GALAHAD NLS

USER DOCUMENTATION GALAHAD Optimization Library version 5.1

1 SUMMARY

This package uses a **regularization method to find a (local) unconstrained minimizer of a differentiable weighted sum-of-squares objective function**

$$f(\mathbf{x}) \stackrel{\rm def}{=} \tfrac{1}{2} \sum_{i=1}^{m} w_i c_i^2(\mathbf{x}) \equiv \tfrac{1}{2} \|\mathbf{c}(\mathbf{x})\|^2_W$$

of many variables x involving positive weights w_i, $i = 1,\ldots,m$. The method offers the choice of direct and iterative solution of the key regularization subproblems, and is most suitable for large problems. First derivatives of the **residual function $\mathbf{c}(\mathbf{x})$** are required, and if second derivatives of the $c_i(\mathbf{x})$ can be calculated, they may be exploited—if suitable products of the first or second derivatives with a vector may be found but not the derivatives themselves, that can also be used to advantage.

ATTRIBUTES — Versions: `GALAHAD_NLS_single`, `GALAHAD_NLS_double`. **Uses:** `GALAHAD_CLOCK`, `GALAHAD_SYMBOLS`, `GALAHAD_NLPT`, `GALAHAD_SPECFILE`, `GALAHAD_PSLS`, `GALAHAD_GLRT`, `GALAHAD_RQS`, `GALAHAD_BSC`, `GALAHAD_SPACE`, `GALAHAD_ROOTS`, `GALAHAD_MOP`, `GALAHAD_NORMS`, `GALAHAD_STRING` and `GALAHAD_BLAS_interface`. **Date:** October 2016. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory. **Language:** Fortran 95 + TR 15581 or Fortran 2003.

2 HOW TO USE THE PACKAGE

The package is available with single, double and (if available) quadruple precision reals, and either 32-bit or 64-bit integers. Access to the 32-bit integer, single precision version requires the `USE` statement

```
USE GALAHAD_NLS_single
```

with the obvious substitution `GALAHAD_NLS_double`, `GALAHAD_NLS_quadruple`, `GALAHAD_NLS_single_64`, `GALAHAD_NLS_double_64` and `GALAHAD_NLS_quadruple_64` for the other variants.

If it is required to use more than one of the modules at the same time, the derived types `SMT_type`, `GALAHAD_userdata_type`, `NLS_time_type`, `NLS_control_type`, `NLS_inform_type`, `NLS_data_type` and `NLPT_problem_type`, (Section 2.4) and the subroutines `NLS_initialize`, `NLS_solve`, `NLS_terminate`, (Section 2.5) and `NLS_read_specfile` (Section 2.9) must be renamed on one of the `USE` statements.

2.1 Basic terminology

The algorithm used is iterative. From the current best estimate of the minimizer $\mathbf{x}_k$, a trial improved point $\mathbf{x}_k + \mathbf{s}_k$ is sought. The correction $\mathbf{s}_k$ is chosen to improve a model $m_k(\mathbf{s})$ of the objective function $f(\mathbf{x}_k + \mathbf{s})$ built around $\mathbf{x}_k$. The model is the sum of two basic components, a suitable approximation $t_k(\mathbf{s})$ of $f(\mathbf{x}_k + \mathbf{s})$, and a regularization term $\frac{\sigma_k}{p} \|\mathbf{s}\|^p_{\mathbf{S}_k}$ involving a weight σ_k, power p and a norm $\|\mathbf{s}\|_{\mathbf{S}_k} \stackrel{\rm def}{=} \sqrt{\mathbf{s}^T \mathbf{S}_k \mathbf{s}}$ for a given positive definite scaling matrix $\mathbf{S}_k$ that is included to prevent large corrections. The weight σ_k is adjusted as the algorithm progresses to ensure convergence.

The model $t_k(\mathbf{s})$ is a truncated Taylor-series approximation, and this relies on being able to compute or estimate derivatives of $\mathbf{c}(\mathbf{x})$. Various models are provided, and each has different derivative requirements. We denote the m by n **residual Jacobian $\mathbf{J}(\mathbf{x})$** as the matrix whose i,j-th component

$$\mathbf{J}(\mathbf{x})_{i,j} \stackrel{\rm def}{=} \partial c_i(\mathbf{x}) / \partial x_j \;\text{ for } i = 1,\ldots,m \text{ and } j = 1,\ldots,n.$$

All use is subject to the conditions of a BSD-3-Clause License.
See `http://galahad.rl.ac.uk/galahad-www/cou.html` **for full details.**

For a given m-vector $\mathbf{y}$, the **weighted residual Hessian** is the sum

$$\mathbf{H}(\mathbf{x},\mathbf{y}) \stackrel{\rm def}{=} \sum_{\ell=1}^{m} y_\ell \mathbf{H}_\ell(\mathbf{x}), \;\; \text{where} \;\; \mathbf{H}_\ell(\mathbf{x})_{i,j} \stackrel{\rm def}{=} \partial^2 c_\ell(\mathbf{x})/\partial x_i \partial x_j \;\; \text{for } i,j = 1,\ldots,n$$

is the Hessian of $c_\ell(\mathbf{x})$. Finally, for a given vector $\mathbf{v}$, we define the **residual-Hessians-vector product matrix**

$$\mathbf{P}(\mathbf{x},\mathbf{v}) \stackrel{\rm def}{=} (\mathbf{H}_1(\mathbf{x})\mathbf{v},\ldots,\mathbf{H}_m(\mathbf{x})\mathbf{v}).$$

The models $t_k(\mathbf{s})$ provided are,

1. the first-order Taylor approximation $f(\mathbf{x}_k)+\mathbf{g}(\mathbf{x}_k)^T\mathbf{s}$, where $\mathbf{g}(\mathbf{x}) = \mathbf{J}^T(\mathbf{x})\mathbf{W}\mathbf{c}(\mathbf{x})$,
2. a barely second-order approximation $f(\mathbf{x}_k)+\mathbf{g}(\mathbf{x}_k)^T\mathbf{s}+\frac{1}{2}\mathbf{s}^T\mathbf{W}\mathbf{s}$,
3. the Gauss-Newton approximation $\frac{1}{2}\|\mathbf{c}(\mathbf{x}_k)+\mathbf{J}(\mathbf{x}_k)\mathbf{s}\|^2_{\mathbf{W}}$,
4. the Newton (second-order Taylor) approximation $f(\mathbf{x}_k)+\mathbf{g}(\mathbf{x}_k)^T s+\frac{1}{2}\mathbf{s}^T[\mathbf{J}^T(\mathbf{x}_k)\mathbf{W}\mathbf{J}(\mathbf{x}_k)+\mathbf{H}(\mathbf{x}_k,\mathbf{W}\mathbf{c}(\mathbf{x}_k))]\mathbf{s}$, and
5. the tensor Gauss-Newton approximation $\frac{1}{2}\|\mathbf{c}(\mathbf{x}_k)+\mathbf{J}(\mathbf{x}_k)\mathbf{s}+\frac{1}{2}\mathbf{s}^T\cdot\mathbf{P}(\mathbf{x}_k,\mathbf{s})\|^2_{\mathbf{W}}$, where the i-th component of $\mathbf{s}^T\cdot\mathbf{P}(\mathbf{x}_k,\mathbf{s})$ is shorthand for the scalar $\mathbf{s}^T\mathbf{H}_i(\mathbf{x}_k)\mathbf{s}$,

where $\mathbf{W}$ is the diagonal matrix of weights w_i, $i = 1,\ldots m$.

Access to a particular model requires that the user is either able to provide the derivatives needed (**"matrix available"**) or that the products of these derivatives (and their transposes) with specified vectors are possible (**"matrix free"**).

2.2 Matrix storage formats

The matrices $\mathbf{J}(\mathbf{x})$, $\mathbf{H}(\mathbf{x},\mathbf{y})$ and $\mathbf{P}(\mathbf{x},\mathbf{v})$ (as required and when available) may be stored in a variety of input formats.

2.2.1 Dense storage format

The matrix $\mathbf{J}$ is stored as a compact dense matrix by rows, that is, the values of the entries of each row in turn are stored in order within an appropriate real one-dimensional array. Component $n*(i-1)+j$ of the storage array `J%val` will hold the value $\mathbf{J}_{i,j}$ for $i = 1,\ldots,m$, $j = 1,\ldots,n$. Since $\mathbf{H}$ is symmetric, only the lower triangular part (that is the part $\mathbf{H}_{ij}$ for $1 \leq j \leq i \leq n$) should be stored. In this case the lower triangle will be stored by rows, that is component $i*(i-1)/2+j$ of the storage array `H%val` will hold the value $\mathbf{H}_{ij}$ (and, by symmetry, $\mathbf{H}_{ji}$) for $1 \leq j \leq i \leq n$. If tensor-Newton models are used, the required matrix $\mathbf{P}$ may be stored as a compact dense matrix by columns, that is, the values of the entries of each column in turn are stored in order within an appropriate real one-dimensional array.

2.2.2 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l-th entry of $\mathbf{J}$, its row index i, column index j and value $\mathbf{J}_{ij}$ are stored in the l-th components of the integer arrays `J%row`, `J%col` and real array `J%val`. The order is unimportant, but the total number of entries `J%ne` is required. The same scheme is applicable to $\mathbf{H}$ (thus requiring integer arrays `H%row`, `H%col`, a real array `H%val`, and an integer value `H%ne`), except that only the entries in the lower triangle should be stored.

All use is subject to the conditions of a BSD-3-Clause License.
See `http://galahad.rl.ac.uk/galahad-www/cou.html` **for full details.**

2.2.3 Sparse row-wise storage format

Again only the nonzero entries are stored, but this time they are ordered so that those in row i appear directly before those in row $i+1$. For the i-th row of $\mathbf{J}$, the i-th component of the integer array `J%ptr` holds the position of the first entry in this row, while `J%ptr` $(m+1)$ holds the total number of entries plus one. The column indices j and values $\mathbf{J}_{ij}$ of the entries in the i-th row are stored in components $l =$ `J%ptr`(i), ...,`J%ptr` $(i+1)-1$ of the integer array `J%col`, and real array `J%val`, respectively. The same scheme is applicable to $\mathbf{H}$ (thus requiring integer arrays `H%ptr`, `H%col`, and a real array `H%val`), except that only the entries in the lower triangle should be stored.

For sparse matrices, this scheme almost always requires less storage than its predecessor.

2.2.4 Sparse column-wise storage format

For the matrix $\mathbf{P}$, once again only the nonzero entries are stored, but this time they are ordered so that those in column j appear directly before those in column $j+1$. For the j-th column of $\mathbf{P}$, the j-th component of the integer array `P%ptr` holds the position of the first entry in this column, while `P%ptr` $(m+1)$ holds the total number of entries plus one. The row indices i and values $\mathbf{P}_{ij}$ of the entries in the j-th column are stored in components $l =$ `P%ptr`(j), ...,`P%ptr` $(j+1)-1$ of the integer array `P%row`, and real array `P%val`, respectively.

2.2.5 Diagonal storage format

If $\mathbf{H}$ is diagonal (i.e., $\mathbf{H}_{ij} = 0$ for all $1 \leq i \neq j \leq n$) only the diagonal entries $\mathbf{H}_{ii}$ for $1 \leq i \leq n$ need be stored, and the first n components of the array `H%val` may be used for the purpose. There is no sensible equivalent for the non-square matrices $\mathbf{J}$ and $\mathbf{P}$.

2.3 Real and integer kinds

We use the terms integer and real to refer to the fortran keywords `REAL(rp_)` and `INTEGER(ip_)`, where `rp_` and `ip_` are the relevant kind values for the real and integer types employed by the particular module in use. The former are equivalent to default `REAL` for the single precision versions, `DOUBLE PRECISION` for the double precision cases and quadruple-precision if 128-bit reals are available, and correspond to `rp_ = real32`, `rp_ = real64` and `rp_ = real128` respectively as defined by the fortran `iso_fortran_env` module. The latter are default (32-bit) and long (64-bit) integers, and correspond to `ip_ = int32` and `ip_ = int64`, respectively, again from the `iso_fortran_env` module.

2.4 The derived data types

Seven derived data types are accessible from the package.

2.4.1 The derived data type for holding matrices

The derived data type `SMT_TYPE` is used to hold the Hessian matrix $\mathbf{H}$ if this is available. The components of `SMT_TYPE` used here are:

`m` is a scalar component of type `INTEGER(ip_)`, that holds the row dimension of the matrix.

`n` is a scalar component of type `INTEGER(ip_)`, that holds the column dimension of the matrix.

`ne` is a scalar variable of type `INTEGER(ip_)`, that holds the number of matrix entries.

`type` is a rank-one allocatable array of type default `CHARACTER`, that is used to indicate the matrix storage scheme used. Its precise length and content depends on the type of matrix to be stored (see §2.4.2).

All use is subject to the conditions of a BSD-3-Clause License.
See `http://galahad.rl.ac.uk/galahad-www/cou.html` **for full details.**

val is a rank-one allocatable array of type REAL(rp_) and dimension at least ne, that holds the values of the entries. Each pair of off-diagonal entries $h_{ij} = h_{ji}$ of the *symmetric* matrix $\mathbf{H}$ is represented as a single entry (see §2.2.1–2.2.5). Any duplicated entries that appear in the sparse co-ordinate or row-wise schemes will be summed.

row is a rank-one allocatable array of type INTEGER(ip_), and dimension at least ne, that may hold the row indices of the entries. (see §2.2.2 and §2.2.4).

col is a rank-one allocatable array of type INTEGER(ip_), and dimension at least ne, that may hold the column indices of the entries (see §2.2.2–2.2.3).

ptr is a rank-one allocatable array of type INTEGER(ip_), and dimension at least n + 1, that may hold the pointers to the first entry in each row (see §2.2.3–2.2.4).

2.4.2 The derived data type for holding the problem

The derived data type NLPT_problem_type is used to hold the problem. The relevant components of NLPT_problem_type are:

n is a scalar variable of type INTEGER(ip_), that holds the number of optimization variables, n.

m is a scalar variable of type INTEGER(ip_), that holds the number of residual functions, m.

X is a rank-one allocatable array of dimension n and type REAL(rp_), that holds the values $\mathbf{x}$ of the optimization variables. The j-th component of X, $j = 1,\ldots,n$, contains x_j.

C is a rank-one allocatable array of dimension m and type REAL(rp_), that holds the residual values $\mathbf{c}(\mathbf{x})$ at the point $\mathbf{x}$. The i-th component of C, $i = 1,\ldots,m$, contains $\mathbf{c}_i(\mathbf{x})$.

G is a rank-one allocatable array of dimension m and type REAL(rp_), that holds the gradient $g(\mathbf{x}) = \nabla_x f(\mathbf{x})$ of the objective function at the point $\mathbf{x}$. The j-th component of G, $i = 1,\ldots,m$, contains $\partial f(\mathbf{x})/\partial x_j$.

J is scalar variable of type SMT_TYPE that holds the Jacobian matrix $\mathbf{J}(\mathbf{x})$ (if it is available). The following components are used here:

J%type is an allocatable array of rank one and type default CHARACTER, that is used to indicate the storage scheme used. If the dense storage scheme (see Section 2.2.1) is used, the first five components of J%type must contain the string DENSE. For the sparse co-ordinate scheme (see Section 2.2.2), the first ten components of J%type must contain the string COORDINATE, and for the sparse row-wise storage scheme (see Section 2.2.3), the first fourteen components of J%type must contain the string SPARSE_BY_ROWS.

For convenience, the procedure SMT_put may be used to allocate sufficient space and insert the required keyword into J%type. For example, if nlp is of derived type NLS_problem_type and involves a Jacobian we wish to store using the co-ordinate scheme, we may simply

```
CALL SMT_put( nlp%J%type, 'COORDINATE' )
```

See the documentation for the GALAHAD package SMT for further details on the use of SMT_put.

J%ne is a scalar variable of type INTEGER(ip_), that holds the number of entries in $\mathbf{J}$ in the sparse co-ordinate storage scheme (see Section 2.2.2). It need not be set for any of the other three schemes.

J%val is a rank-one allocatable array of type REAL(rp_), that holds the values of the entries of the Jacobian matrix $\mathbf{J}$ in any of the storage schemes discussed in Section 2.2.

J%row is a rank-one allocatable array of type INTEGER(ip_), that holds the row indices of $\mathbf{J}$ in the sparse co-ordinate storage scheme (see Section 2.2.2). It need not be allocated for any of the other three schemes.

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

J%col is a rank-one allocatable array variable of type INTEGER(ip_), that holds the column indices of **J** in either the sparse co-ordinate (see Section 2.2.2), or the sparse row-wise (see Section 2.2.3) storage scheme. It need not be allocated when the dense or diagonal storage schemes are used.

J%ptr is a rank-one allocatable array of dimension n+1 and type INTEGER(ip_), that holds the starting position of each row of **J**, as well as the total number of entries plus one, in the sparse row-wise storage scheme (see Section 2.2.3). It need not be allocated when the other schemes are used.

H is scalar variable of type SMT_TYPE that may hold the weighted Hessian matrix $\mathbf{H}(\mathbf{x},\mathbf{y})$ (if it is available). The following components are used here:

H%type is an allocatable array of rank one and type default CHARACTER, that is used to indicate the storage scheme used. If the dense storage scheme (see Section 2.2.1) is used, the first five components of H%type must contain the string DENSE. For the sparse co-ordinate scheme (see Section 2.2.2), the first ten components of H%type must contain the string COORDINATE, for the sparse row-wise storage scheme (see Section 2.2.3), the first fourteen components of H%type must contain the string SPARSE_BY_ROWS, and for the diagonal storage scheme (see Section 2.2.5), the first eight components of H%type must contain the string DIAGONAL.

For convenience, the procedure SMT_put may be used to allocate sufficient space and insert the required keyword into H%type. For example, if nlp is of derived type NLS_problem_type and involves a Hessian we wish to store using the co-ordinate scheme, we may simply

```
CALL SMT_put( nlp%H%type, 'COORDINATE' )
```

See the documentation for the GALAHAD package SMT for further details on the use of SMT_put.

H%ne is a scalar variable of type INTEGER(ip_), that holds the number of entries in the **lower triangular** part of **H** in the sparse co-ordinate storage scheme (see Section 2.2.2). It need not be set for any of the other three schemes.

H%val is a rank-one allocatable array of type REAL(rp_), that holds the values of the entries of the **lower triangular** part of the Hessian matrix **H** in any of the storage schemes discussed in Section 2.2.

H%row is a rank-one allocatable array of type INTEGER(ip_), that holds the row indices of the **lower triangular** part of **H** in the sparse co-ordinate storage scheme (see Section 2.2.2). It need not be allocated for any of the other three schemes.

H%col is a rank-one allocatable array variable of type INTEGER(ip_), that holds the column indices of the **lower triangular** part of **H** in either the sparse co-ordinate (see Section 2.2.2), or the sparse row-wise (see Section 2.2.3) storage scheme. It need not be allocated when the dense or diagonal storage schemes are used.

H%ptr is a rank-one allocatable array of dimension n+1 and type INTEGER(ip_), that holds the starting position of each row of the **lower triangular** part of **H**, as well as the total number of entries plus one, in the sparse row-wise storage scheme (see Section 2.2.3). It need not be allocated when the other schemes are used.

P is scalar variable of type SMT_TYPE that may hold the matrix of residual-Hessians-vector products $\mathbf{P}(\mathbf{x},\mathbf{v}) = (\mathbf{H}_1(\mathbf{x})\mathbf{v},\ldots,\mathbf{H}_m(\mathbf{x})\mathbf{v})$ for a supplied vector **v** (if it is needed). The matrix **P** is held as a sparse matrix by columns (see Section 2.2.4) or as a dense matrix (see Section 2.2.1). The following components are used here:

P%type is an allocatable array of rank one and type default CHARACTER, that is used to indicate the storage scheme used. If the dense storage scheme (see Section 2.2.1) is used, the first sixteen components of P%type must contain the string DENSE_BY_COLUMNS, while and for the sparse column-wise storage scheme (see Section 2.2.4), the first seventeen components of P%type must contain the string SPARSE_BY_COLUMNS.

For convenience, the procedure SMT_put may be used to allocate sufficient space and insert the required keyword into P%type. For example, if nlp is of derived type NLS_problem_type and involves a residual=Hessian-vector product matrix we wish to store using the dense scheme, we may simply

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

```
CALL SMT_put( nlp%P%type, 'DENSE_BY_COLUMNS' )
```

See the documentation for the GALAHAD package SMT for further details on the use of SMT_put.

P%val is a rank-one allocatable array of type REAL(rp_), that holds the values of the entries of the product matrix **P** stored as consecutive columns.

P%row is a rank-one allocatable array of type INTEGER(ip_), that holds the row indices of **P** in the same order as used for P%val.

P%ptr is a rank-one allocatable array of dimension m+1 and type INTEGER(ip_), that holds the starting position of each column of **P**, as well as the total number of entries plus one.

pname is a scalar variable of type default CHARACTER and length 10, which contains the "name" of the problem for printing. The default "empty" string is provided.

VNAMES is a rank-one allocatable array of dimension n and type default CHARACTER and length 10, whose j-th entry contains the "name" of the j-th variable for printing. This is only used if "debug"printing control%print_level > 4) is requested, and will be ignored if the array is not allocated.

CNAMES is a rank-one allocatable array of dimension m and type default CHARACTER and length 10, whose i-th entry contains the "name" of the i-th residual for printing. This is only used if "debug"printing control%print_level > 4) is requested, and will be ignored if the array is not allocated.

2.4.3 The derived data type for holding control parameters

The derived data type NLS_control_type is used to hold controlling data. Default values may be obtained by calling NLS_initialize (see Section 2.5.1), while components may also be changed by calling GALAHAD_NLS_read_spec (see Section 2.9.1). The derived type NLS_subproblem_control_type comprises all of the components of NLS_control_type except for the last (i.e., subproblem_control). The components of NLS_control_type are:

error is a scalar variable of type INTEGER(ip_), that holds the stream number for error messages. Printing of error messages in NLS_solve and NLS_terminate is suppressed if error ≤ 0. The default is error = 6.

out is a scalar variable of type INTEGER(ip_), that holds the stream number for informational messages. Printing of informational messages in NLS_solve is suppressed if out < 0. The default is out = 6.

print_level is a scalar variable of type INTEGER(ip_), that is used to control the amount of informational output which is required. No informational output will occur if print_level ≤ 0. If print_level $= 1$, a single line of output will be produced for each iteration of the process. If print_level ≥ 2, this output will be increased to provide significant detail of each iteration. The default is print_level = 0.

maxit is a scalar variable of type INTEGER(ip_), that holds the maximum number of iterations which will be allowed in NLS_solve. The default is maxit = 1000.

start_print is a scalar variable of type INTEGER(ip_), that specifies the first iteration for which printing will occur in NLS_solve. If start_print is negative, printing will occur from the outset. The default is start_print = -1.

stop_print is a scalar variable of type INTEGER(ip_), that specifies the last iteration for which printing will occur in NLS_solve. If stop_print is negative, printing will occur once it has been started by start_print. The default is stop_print = -1.

print_gap is a scalar variable of type INTEGER(ip_). Once printing has been started, output will occur once every print_gap iterations. If print_gap is no larger than 1, printing will be permitted on every iteration. The default is print_gap = 1.

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

alive_unit is a scalar variable of type INTEGER(ip_). If alive_unit > 0, a temporary file named alive_file (see below) will be created on stream number alive_unit on initial entry to GALAHAD_NLS_solve, and execution of GALAHAD_NLS_solve will continue so long as this file continues to exist. Thus, a user may terminate execution simply by removing the temporary file from this unit. If alive_unit ≤ 0, no temporary file will be created, and execution cannot be terminated in this way. The default is alive_unit $= 40$.

jacobian_available is a scalar variable of type INTEGER(ip_), that specifies the availability of the residual Jacobian. Possible values are:

- $\geq$ 2 The Jacobian is available.
- 1 The Jacobian is not available explicitly but its effect may be accessed by matrix-vector products, i.e., it is "matrix-free".
- $\leq$ 0 The Jacobian is not available either explicitly or via matrix-vector products.

The default is jacobian_available = 1.

hessian_available is a scalar variable of type INTEGER(ip_), that specifies the availability of the weighted-residual Hessian. Possible values are:

- $\geq$ 2 The Hessian is available.
- 1 The Hessian is not available explicitly but its effect may be accessed by matrix-vector products, i.e., it is "matrix-free".
- $\leq$ 0 The Hessian is not available either explicitly or via matrix-vector products.

The default is hessian_available = 0.

model is a scalar variable of type INTEGER(ip_), that specifies which model to be used to approximate $f(\mathbf{x})$ when computing the step. Possible values are:

- $\leq$ 0 the model is chosen automatically on the basis of which option looks likely to be the most efficient at any given stage of the solution process. Different models may be used at different stages. **Not yet implemented.**
- 1 a first-order model, not involving the Hessian, will be used.
- 2 a barely-second-order model, in which the Hessian is approximated by the matrix $\mathbf{W}$, will be used.
- 3 a Gauss-Newton model, in which the Hessian of $f(\mathbf{x})$ is approximated by $\mathbf{J}^T(\mathbf{x})\mathbf{W}\mathbf{J}(\mathbf{x})$, will be used.
- 4 a second-order Newton model, in which the exact Hessian of $f(\mathbf{x})$, $\mathbf{J}^T(\mathbf{x})\mathbf{W}\mathbf{J}(\mathbf{x}) + \mathbf{H}(\mathbf{x}, \mathbf{W}\mathbf{c}(\mathbf{x}))$, will be used.
- 5 an adaptive second-order model, in which there is a transition from Gauss-Newton to Newton models, will be used.
- 6 a tensor Gauss-Newton model will be used, and an approximate minimizer of this model will be found by a Gauss-Newton method.
- 7 a tensor Gauss-Newton model will be used, and an approximate minimizer of this model will be found by a Newton method.
- $\geq$ 8 a tensor Gauss-Newton model will be used, and an approximate minimizer of this model will be found by a method that adapts from Gauss-Newton to Newton.

See §2.1 for further details. The default is model = 3.

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

norm is a scalar variable of type INTEGER(ip_), that specifies which norm is to be used to define the regularization. In particular the norm $\|\cdot\|$ will be defined by a symmetric, positive-definite matrix $\mathbf{S}(\mathbf{x})$ that may depend on $\mathbf{x}$ so that for every vector $\mathbf{v}$, $\|\mathbf{v}\|^2 = \mathbf{v}^T\mathbf{S}(\mathbf{x})\mathbf{v}$. If %subproblem_direct = .FALSE., the same $\mathbf{S}(\mathbf{s})$ also defines the preconditioner to be used to accelerate the generalized-Lanczos inner model minimization. Possible values are:

- -3 the user's own norm will be used.
- -2 a norm based on a limited-memory BFGS formula will be used. **Not yet implemented.**
- -1 the Euclidean (ℓ_2-) norm is used.
- 0 the type is chosen automatically on the basis of which option looks likely to be the most efficient at any given stage of the solution process. Different norms may be used at different stages. **Not yet implemented.**
- 1 $\mathbf{S}$ is the diagonal of the matrix $\mathbf{J}^T(\mathbf{x}_k)\mathbf{W}\mathbf{J}(\mathbf{x}_k)$, or equivalently the squares of the two-norms of the columns of $\mathbf{W}^{\frac{1}{2}}\mathbf{J}(\mathbf{x}_k)$.
- 2 $\mathbf{S}$ is the diagonal of the Hessian matrix, $\mathbf{J}^T(\mathbf{x}_k)\mathbf{W}\mathbf{J}(\mathbf{x}_k) + \mathbf{H}(\mathbf{x}_k, \mathbf{W}\mathbf{c}(\mathbf{x}_k))$, suitably modified to ensure that it is significantly positive definite.
- 3 $\mathbf{S}$ is the Hessian matrix whose entries outside a band of given semi-bandwidth are replaced by zeros (see control%PSLS_control%semi_bandwidth below).
- 4 $\mathbf{S}$ is the Hessian matrix whose entries outside a bandwidth-reduced reordered band of given semi-bandwidth are replaced by zeros (see control%PSLS_control%semi_bandwidth below).
- 5 $\mathbf{S}$ is the (possibly perturbed) Hessian, using the Schnabel-Eskow modification method to ensure that the resultant matrix is positive definite.
- 6 $\mathbf{S}$ is the (possibly perturbed) Hessian, using the Gill-Murray-Poncéleon-Saunders modification method to ensure that the resultant matrix is positive definite.
- 7 $\mathbf{S}$ will be that from the incomplete factorization of the Hessian using the Lin-Moré method (see control%PSLS_control%icfs_vectors below).
- 8 $\mathbf{S}$ will be that from the incomplete factorization of the Hessian using the method implemented by HSL_MI28.

Any value outside this range, and those not yet implemented, will be treated as the default, norm = 1.

non_monotone is a scalar variable of type INTEGER(ip_), that specifies the history-length for non-monotone descent strategy. Any non-positive value results in standard monotone descent, for which merit function improvement occurs at each iteration. There are often definite advantages in using a non-monotone strategy with a modest history, since the occasional local increase in the merit function may enable the algorithm to move across (gentle) "ripples" in the merit function surface. However, we do not usually recommend large values of non_monotone. The default is non_monotone = 1.

weight_update_strategy is a scalar variable of type INTEGER(ip_), that specifies the way in which the regularization weight will be adjusted at the end of each iteration. Possible values are:

- 1 the traditional acceptance and rejection strategy as described below.
- 2 the traditional strategy, except that a zero weight will be tried first after a very successful step.
- 3 a more sophisticated strategy that mimics that proposed for trust-region methods by Gould, Porcelli and Toint.

Any other value will be considered as if weight_update_strategy = 1, and this is the default.

stop_c_absolute is a scalar variable of type REAL(rp_), that is used to specify the maximum permitted two-norm of the residual, $\|\mathbf{c}(\mathbf{x})\|_{\mathbf{W}}$, (see Section 4) at the estimate of the solution sought. Any negative value will be recorded as zero. The default is stop_c_absolute = 10^{-6}.

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

stop_c_relative is a scalar variable of type REAL(rp_), that is used to specify the largest relative reduction in the two-norm of the residual, $\|\mathbf{c}(\mathbf{x})\|_{\mathbf{W}}$, that will be permitted (see Section 4) at the estimate of the solution sought compared to that at the initial point. Any negative value will be recorded as zero. The default is stop_c_relative = 0.0.

stop_g_absolute is a scalar variable of type REAL(rp_), that is used to specify the maximum permitted two-norm of the gradient, $\|\mathbf{J}^T(\mathbf{x})\mathbf{W}\mathbf{c}(\mathbf{x})\|_2/\|\mathbf{c}(\mathbf{x})\|_{\mathbf{W}}$, of the residual $\|\mathbf{c}(\mathbf{x})\|_{\mathbf{W}}$, (see Section 4) at the estimate of the solution sought. Any negative value will be recorded as zero. The default is stop_g_absolute = 10^{-6}.

stop_g_relative is a scalar variable of type REAL(rp_), that is used to specify the largest relative reduction in the norm of the gradient of the residual that will be permitted (see Section 4) at the estimate of the solution sought compared to that at the initial point. Any negative value will be recorded as zero. The default is stop_g_relative = 0.0.

stop_s is a scalar variable of type REAL(rp_), that is used to specify the minimum acceptable correction step $\mathbf{s}$ relative to the current estimate of the solution $\mathbf{x}$ The algorithm will be deemed to have converged if $|s_i| \leq$ stop_s $*\max(1,|x_i|)$ for all $i = 1,\ldots,n$. The default is stop_s = u, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_NLS_double).

regularization_power is a scalar variable of type REAL(rp_), that holds the required initial value of the regularization weight. If regularization_power ≤ 2.0, the weight will be chosen automatically by GALAHAD_NLS_solve. The default is regularization_power = -1.0.

initial_weight is a scalar variable of type REAL(rp_), that holds the required initial value of the regularization weight. If initial_weight ≤ 0, the weight will be chosen automatically by GALAHAD_NLS_solve. The default is initial_weight = 100.0.

minimum_weight is a scalar variable of type REAL(rp_), that holds the largest permitted value of the regularization weight as the algorithm proceeds. The default is minimum_weight = 10^{-8}.

initial_inner_weight is a scalar variable of type REAL(rp_), that holds the required initial value of the regularization weight if the model minimization itself requires an inner iteration (i.e., if %model ≥ 6). If initial_inner_weight ≤ 0, the weight will be chosen automatically by GALAHAD_NLS_solve. The default is initial_inner_weight = 0.0.

eta_successful, eta_very_successful and eta_too_successful are scalar variables of type default REAL(rp_), that control the acceptance and rejection of the trial step and the updates to the regularization weight. At every iteration, the ratio of the actual reduction in the merit function following the trial step to that predicted by the model is computed. The step is accepted whenever this ratio exceeds eta_successful; otherwise the regularization weight will be reduced. If, in addition, the ratio exceeds eta_very_successful but not eta_too_successful, the regularization weight may be increased. The defaults are eta_successful = 10^{-8}, eta_very_successful = 0.9 and eta_too_successful = 2.0.

weight_increase, weight_decrease, weight_increase_max and weight_decrease_min are scalar variables of type REAL(rp_), that control the maximum amounts by which the regularization weight can contract or expand during an iteration. The weight will be decreased by powers of weight_decrease, but not in total more than weight_decrease_min, until it is smaller than the norm of the current step. It can be increased by at most a factor weight_increase, but not in total less than weight_increase_max. The defaults are weight_increase = 10.0, weight_decrease = 0.1, weight_increase_max = 100.0 and weight_decrease_min = 0.1.

switch_to_newton is a scalar variable of type REAL(rp_), that is used to specify the value of the two-norm of the gradient, $\|\mathbf{J}^T(\mathbf{x})\mathbf{W}\mathbf{c}(\mathbf{x})\|_2/\|\mathbf{c}(\mathbf{x})\|_{\mathbf{W}}$, of the residual $\|\mathbf{c}(\mathbf{x})\|_{\mathbf{W}}$, required before a switch is made from the Gauss-Newton model to the Newton one when %model = 5). The default is switch_to_newton = 0.1.

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

cpu_time_limit is a scalar variable of type REAL(rp_), that is used to specify the maximum permitted CPU time. Any negative value indicates no limit will be imposed. The default is cpu_time_limit = - 1.0.

clock_time_limit is a scalar variable of type REAL(rp_), that is used to specify the maximum permitted elapsed system clock time. Any negative value indicates no limit will be imposed. The default is clock_time_limit = - 1.0.

subproblem_direct is a scalar variable of type default LOGICAL, that should be set .TRUE. if a direct (factorization) method is desired when solving for the step, and .FALSE. if an iterative method suffices. The default is subproblem_direct = .FALSE..

renormalize_weight is a scalar variable of type default LOGICAL, that should be set .TRUE. if the regularization weight is to be re-normaized to account for the shape of the regularization norm every iteration, and .FALSE. if no re-normalization is required. The default is renormalize_weight = .FALSE..

magic_step is a scalar variable of type default LOGICAL, that should be set .TRUE. if additional "magic" steps are to be used in order to improve the objective as the iteration proceeds, and .FALSE. if no "magic" steps are used. The default is magic_step = .FALSE..

print_obj is a scalar variable of type default LOGICAL, that should be set .TRUE. if output values relate to $f(\mathbf{x})$, and .FALSE. if they relate to $\|\mathbf{c}(\mathbf{x})\|_{\mathbf{W}}$. The default is print_obj = .FALSE..

space_critical is a scalar variable of type default LOGICAL, that must be set .TRUE. if space is critical when allocating arrays and .FALSE. otherwise. The package may run faster if space_critical is .FALSE. but at the possible expense of a larger storage requirement. The default is space_critical = .FALSE..

deallocate_error_fatal is a scalar variable of type default LOGICAL, that must be set .TRUE. if the user wishes to terminate execution if a deallocation fails, and .FALSE. if an attempt to continue will be made. The default is deallocate_error_fatal = .FALSE..

alive_file is a scalar variable of type default CHARACTER and length 30, that gives the name of the temporary file whose removal from stream number alive_unit terminates execution of GALAHAD_NLS_solve. The default is alive_unit = ALIVE.d.

prefix is a scalar variable of type default CHARACTER and length 30, that may be used to provide a user-selected character string to preface every line of printed output. Specifically, each line of output will be prefaced by the string prefix(2:LEN(TRIM(prefix))-1), thus ignoreing the first and last non-null components of the supplied string. If the user does not want to preface lines by such a string, they may use the default prefix = "".

RQS_control is a scalar variable of type RQS_control_type whose components are used to control the direct regularization step calculation (if any), performed by the package GALAHAD_RQS. See the specification sheet for the package GALAHAD_RQS for details, and appropriate default values (but note that values of RQS_control%initial_multiplier and RQS_control%new_h may be changed by GALAHAD_NLS_solve).

GLRT_control is a scalar variable of type GLRT_control_type whose components are used to control the iterative regularization step calculation (if any), performed by the package GALAHAD_GLRT. See the specification sheet for the package GALAHAD_GLRT for details, and appropriate default values (but note that value of GLRT_control%unitm may be changed by GALAHAD_NLS_solve).

PSLS_control is a scalar variable of type PSLS_control_type whose components are used to control the preconditioning aspects of the calculation, as performed by the package GALAHAD_PSLS. See the specification sheet for the package GALAHAD_PSLS for details, and appropriate default values (but note that values for PSLS_control%preconditioner, PSLS_control%semi_bandwidth and PSLS_control%icfs_vectors may be overridden by GALAHAD_NLS_solve).

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

BSC_control is a scalar variable of type BSC_control_type whose components are used to control the assembly of the Schur complement, if required, as performed by the package GALAHAD_BSC. See the specification sheet for the package GALAHAD_BSC for details, and appropriate default values.

ROOTS_control is a scalar variable of type ROOTS_control_type whose components are used to control polynomial root-finding methods needed, as performed by the package GALAHAD_ROOTS. See the specification sheet for the package GALAHAD_ROOTS for details, and appropriate default values.

subproblem_control is a scalar variable of type NLS_subproblem_control_type whose components are used to control the solution of the least-squares subproblem that defines the step. The only differences are that the default values subproblem_control%maxit = 50 and subproblem_control%print_obj = .TRUE..

2.4.4 The derived data type for holding timing information

The derived data type NLS_time_type is used to hold elapsed CPU and system clock times for the various parts of the calculation. The components of NLS_time_type are:

total is a scalar variable of type default REAL, that gives the CPU total time spent in the package.

preprocess is a scalar variable of type REAL(rp_), that gives the CPU time spent reordering the problem to standard form prior to solution.

analyse is a scalar variable of type REAL(rp_), that gives the CPU time spent analysing required matrices prior to factorization.

factorize is a scalar variable of type REAL(rp_), that gives the CPU time spent factorizing the required matrices.

solve is a scalar variable of type REAL(rp_), that gives the CPU time spent using the factors to solve relevant linear equations.

clock_total is a scalar variable of type default REAL, that gives the total elapsed system clock time spent in the package.

clock_preprocess is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent reordering the problem to standard form prior to solution.

clock_analyse is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent analysing required matrices prior to factorization.

clock_factorize is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent factorizing the required matrices.

clock_solve is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent using the factors to solve relevant linear equations.

2.4.5 The derived data type for holding informational parameters

The derived data type NLS_inform_type is used to hold parameters that give information about the progress and needs of the algorithm. The derived type NLS_subproblem_inform_type comprises all of the components of NLS_inform_type except for the last (i.e., subproblem_inform). The components of NLS_inform_type are:

status is a scalar variable of type INTEGER(ip_), that gives the exit status of the algorithm. See Sections 2.7 and 2.8 for details.

alloc_status is a scalar variable of type INTEGER(ip_), that gives the status of the last attempted array allocation or deallocation. This will be 0 if status = 0.

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

bad_alloc is a scalar variable of type default CHARACTER and length 80, that gives the name of the last internal array for which there were allocation or deallocation errors. This will be the null string if status = 0.

iter is a scalar variable of type INTEGER(ip_), that holds the number of iterations performed.

cg_iter is a scalar variable of type INTEGER(ip_), that gives the total number of conjugate-gradient iterations required.

c_eval is a scalar variable of type INTEGER(ip_), that gives the total number of residual function evaluations performed.

j_eval is a scalar variable of type INTEGER(ip_), that gives the total number of residual Jacobian evaluations performed.

h_eval is a scalar variable of type INTEGER(ip_), that gives the total number of weighted Hessian evaluations performed.

factorization_max is a scalar variable of type INTEGER(ip_), that gives the largest number of factorizations required during a subproblem solution.

factorization_status is a scalar variable of type INTEGER(ip_), that gives the return status from the matrix factorization.

max_entries_factors is a scalar variable of type INTEGER(int64), that gives the maximum number of entries in any of the matrix factorizations performed during the calculation.

factorization_integer is a scalar variable of type default INTEGER(ip_), that gives the amount of integer storage used for the matrix factorization.

factorization_real is a scalar variable of type INTEGER(ip_), that gives the amount of real storage used for the matrix factorization.

factorization_average is a scalar variable of type REAL(rp_), that gives the average number of factorizations per subproblem solved.

obj is a scalar variable of type REAL(rp_), that holds the value of the objective function $f(\mathbf{x})$ at the best estimate of the solution found.

norm_c is a scalar variable of type REAL(rp_), that holds the value of the two-norm of the residual function, $\|\mathbf{c}(\mathbf{x})\|_{\mathbf{W}}$. at the best estimate of the solution found.

norm_g is a scalar variable of type REAL(rp_), that holds the value of the norm of the gradient of the two-norm of the residual function, $\|\mathbf{J}^T(\mathbf{x})\mathbf{W}\mathbf{c}(\mathbf{x})\|_2/\|\mathbf{c}(\mathbf{x})\|_{\mathbf{W}}$, at the best estimate of the solution found.

weight is a scalar variable of type REAL(rp_), that holds the final value of the regularization weight used.

time is a scalar variable of type NLS_time_type whose components are used to hold elapsed elapsed CPU and system clock times for the various parts of the calculation (see Section 2.4.4).

RQS_inform is a scalar variable of type RQS_inform_type whose components give information about the progress and needs of the direct solution stages of the algorithm performed by the package GALAHAD_RQS. See the specification sheet for the package GALAHAD_RQS for details.

GLRT_inform is a scalar variable of type GLRT_inform_type whose components give information about the progress and needs of the iterative solution stages of the algorithm performed by the package GALAHAD_GLRT. See the specification sheet for the package GALAHAD_GLRT for details.

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

PSLS_inform is a scalar variable of type PSLS_inform_type whose components give information about the progress and needs of the preconditioning stages of the algorithm performed by the package GALAHAD_PSLS. See the specification sheet for the package GALAHAD_PSLS for details.

BSC_inform is a scalar variable of type BSC_inform_type whose components give information about the progress and needs of the construction of required Schur complements as performed by the package GALAHAD_BSC. See the specification sheet for the package GALAHAD_BSC for details.

ROOTS_inform is a scalar variable of type ROOTS_inform_type whose components give information about the progress and needs of the required polynomial root-find calculations as performed by the package GALAHAD_ROOTS. See the specification sheet for the package GALAHAD_ROOTS for details.

subproblem_inform is a scalar variable of type NLS_subproblem_inform_type whose components are used to give information about the solution of the least-squares subproblem that defines the step.

2.4.6 The derived data type for holding problem data

The derived data type NLS_data_type is used to hold all the data for a particular problem, or sequences of problems with the same structure, between calls of NLS procedures. This data should be preserved, untouched (except as directed on return from GALAHAD_NLS_solve with positive values of inform%status, see Section 2.7), from the initial call to NLS_initialize to the final call to NLS_terminate.

2.4.7 The derived data type for holding user data

The derived data type GALAHAD_userdata_type is available from the package GALAHAD_userdata to allow the user to pass data to and from user-supplied subroutines for function and derivative calculations (see Section 2.6). Components of variables of type GALAHAD_userdata_type may be allocated as necessary. The following components are available:

integer is a rank-one allocatable array of type INTEGER(ip_).

real is a rank-one allocatable array of type default REAL(rp_)

complex is a rank-one allocatable array of type default COMPLEX (double precision complex in GALAHAD_NLS_double).

character is a rank-one allocatable array of type default CHARACTER.

logical is a rank-one allocatable array of type default LOGICAL.

integer_pointer is a rank-one pointer array of type INTEGER(ip_).

real_pointer is a rank-one pointer array of type default REAL(rp_)

complex_pointer is a rank-one pointer array of type default COMPLEX (double precision complex in GALAHAD_NLS_-double).

character_pointer is a rank-one pointer array of type default CHARACTER.

logical_pointer is a rank-one pointer array of type default LOGICAL.

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

2.5 Argument lists and calling sequences

There are three procedures for user calls (see Section 2.9 for further features):

1. The subroutine NLS_initialize is used to set default values, and initialize private data, before solving one or more problems with the same sparsity and bound structure.
2. The subroutine NLS_solve is called to solve the problem.
3. The subroutine NLS_terminate is provided to allow the user to automatically deallocate array components of the private data, allocated by NLS_solve, at the end of the solution process. It is important to do this if the data object is re-used for another problem **with a different structure** since NLS_initialize cannot test for this situation, and any existing associated targets will subsequently become unreachable.

We use square brackets [ ] to indicate OPTIONAL arguments.

2.5.1 The initialization subroutine

Default values are provided as follows:

```
CALL NLS_initialize( data, control, inform )
```

data is a scalar INTENT(INOUT) argument of type NLS_data_type (see Section 2.4.6). It is used to hold data about the problem being solved.

control is a scalar INTENT(OUT) argument of type NLS_control_type (see Section 2.4.3). On exit, control contains default values for the components as described in Section 2.4.3, while control%subproblem_control contains default values that are required for the model minimization if control%model $\geq$ 6). These values should only be changed after calling NLS_initialize.

inform is a scalar INTENT(OUT) argument of type NLS_inform_type (see Section 2.4.5). A successful call to NLS_initialize is indicated when the component inform%status has the value 0. For other return values of inform%status, see Section 2.8. The components of inform correspond to the main algorithm, while those in inform%subproblem_inform refer to the model minimization if control%model $\geq$ 6).

2.5.2 The minimization subroutine

The minimization algorithm is called as follows:

```
CALL NLS_solve( nlp, control, inform, data, userdata[, W, eval_C, eval_J, eval_H,    &
                eval_JPROD, eval_HPROD, eval_HPRODS, eval_SCALE ] )
```

nlp is a scalar INTENT(INOUT) argument of type NLPT_problem_type (see Section 2.4.2). It is used to hold data about the problem being solved. For a new problem, the user must allocate all the array components, and set values for nlp%n and the required non-real components of nlp%J, nlp%H and nlp%P depending on what level of derivatives are required by the model requested. Specifically, depending on control parameters in control below,

if control%jacobian_available = 2 and control%model $\geq$ 3, all appropriate components of nlp%J should be allocated and, with the exception of nlp%J%val, filled with data,

if control%hessian_available = 2 and control%model $\geq$ 4, all appropriate components of nlp%H should be allocated and, with the exception of nlp%H%val, filled with data, and

if control%model $\geq$ 6, all appropriate components of nlp%P should be allocated and, with the exception of nlp%P%val, filled with data.

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

Users are free to choose whichever of the matrix formats described in Section 2.2 is appropriate for **J** and **H** for their application, but **P** must be input by columns.

The component nlp%X must be set to an initial estimate, $\mathbf{x}^0$, of the minimization variables. A good choice will increase the speed of the package, but the underlying method is designed to converge (at least to a local solution) from an arbitrary initial guess.

On exit, the component nlp%X will contain the best estimates of the minimization variables **x**.

Restrictions: nlp%n >0, nlp%m >0, nlp%J%type $\in$ {'DENSE','COORDINATE','SPARSE_BY_ROWS'} and nlp%H%type $\in$ {'DENSE','COORDINATE','SPARSE_BY_ROWS','DIAGONAL'}.

control is a scalar INTENT(IN) argument of type NLS_control_type (see Section 2.4.3). Default values may be assigned by calling NLS_initialize prior to the first call to NLS_solve. The argument control%PSLS_control%preconditioner will be overridden by control%norm.

The function and derivative requirements are governed by the value of control%model. The precise needs are specified in Table 2.1.

		matrix available (eval_/status)					matrix free (eval_/status)				
%model	C/2	J/3	H/4	JPROD/5	HPROD/6	HPRODS/7	J/3	H/4	JPROD/5	HPROD/6	HPRODS/7
1 & 2	√			√					√		
3	√	√							√		
4 & 5	√	√	√						√	√	
6 & 7	√	√	√			√			√	√	√

Table 2.1: Evaluation requirements for models supported. Key: A √ in box in column A/n means that the external subroutine eval_A will be used if provided, but otherwise that reverse communication will be invoked with inform%status = n. No provision for a derivative need be made for empty boxes, and thus no eval_A need be present for these cases, nor will returns with inform%status = n occur.

inform is a scalar INTENT(INOUT) argument of type NLS_inform_type (see Section 2.4.5). On initial entry, the component inform%status must be set to the value 1. Other entries need note be set. A successful call to NLS_solve is indicated when the component inform%status has the value 0. For other return values of inform%status, see Sections 2.7 and 2.8.

data is a scalar INTENT(INOUT) argument of type NLS_data_type (see Section 2.4.6). It is used to hold data about the problem being solved. With the possible exceptions of the components data%eval_status and data%U (see Section 2.7), it must not have been altered **by the user** since the last call to NLS_initialize.

userdata is a scalar INTENT(INOUT) argument of type GALAHAD_userdata_type whose components may be used to communicate user-supplied data to and from the OPTIONAL subroutines eval_C, eval_J, eval_H, eval_JPROD, eval_HPROD, eval_HPRODS and eval_SCALE (see Section 2.4.7).

W is an OPTIONAL rank-one array of type REAL(rp_) whose components specifies the diagonal weighting matrix **W** that defines the objective function $f(\mathbf{x})$. If W is present, it must be of length at least n, and W(i) should contain $w_i > 0$, $i = 1,\ldots,n$. If **W** is absent, the weights w_i will all be taken to be 1.0.

eval_C is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the residual function $\mathbf{c}(\mathbf{x})$ at a given vector **x**. See Section 2.6.1 for details. If eval_C is present, it must be declared EXTERNAL in the calling program. If eval_C is absent, GALAHAD_NLS_solve will use reverse communication to obtain objective function values (see Section 2.7).

eval_J is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the Jacobian of the residual function $\mathbf{J}(\mathbf{x})$ at a given vector **x**. See Section 2.6.2 for details, but first check Table 2.1 to see if the subroutine is not needed. If eval_J is present, it must be declared EXTERNAL in the calling program. If eval_J is absent, GALAHAD_NLS_solve will use reverse communication to obtain gradient values (see Section 2.7).

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

eval_H is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the weighted residual Hessian $\mathbf{H}(\mathbf{x},\mathbf{y})$ at a given vector pair $(\mathbf{x},\mathbf{y})$. See Section 2.6.3 for details, but first check Table 2.1 to see if the subroutine is not needed. If eval_H is present, it must be declared EXTERNAL in the calling program. If eval_H is absent, GALAHAD_NLS_solve will use reverse communication to obtain Hessian function values (see Section 2.7).

eval_JPROD is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of sum $\mathbf{u}+\mathbf{J}(\mathbf{x})\mathbf{v}$ or $\mathbf{u}+\mathbf{J}^T(\mathbf{x})\mathbf{v}$ involving the product between the Jacobian of the residual function $\mathbf{J}(\mathbf{x})$ or its transpose with a given vector $\mathbf{v}$. See Section 2.6.4 for details, but first check Table 2.1 to see if the subroutine is not needed. If eval_JPROD is present, it must be declared EXTERNAL in the calling program. If eval_JPROD is absent, GALAHAD_NLS_solve will use reverse communication to obtain Jacobian-vector products (see Section 2.7).

eval_HPROD is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of sum $\mathbf{u}+\mathbf{H}(\mathbf{x},\mathbf{y})\mathbf{v}$ involving the product of the weighted residual Hessian $\mathbf{H}(\mathbf{x},\mathbf{y})$ at a given vector pair $(\mathbf{x},\mathbf{y})$ with a given vector $\mathbf{v}$. See Section 2.6.5 for details, but first check Table 2.1 to see if the subroutine is not needed. If eval_HPROD is present, it must be declared EXTERNAL in the calling program. If eval_HPROD is absent, GALAHAD_NLS_solve will use reverse communication to obtain Hessian-vector products (see Section 2.7).

eval_HPRODS is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the values of the residual-Hessians-vector product matrix, $\mathbf{P}(\mathbf{x},\mathbf{v})$ at a given vector pair $(\mathbf{x},\mathbf{v})$. See Section 2.6.6 for details, but first check Table 2.1 to see if the subroutine is not needed. If eval_HPRODS is present, it must be declared EXTERNAL in the calling program. If eval_HPRODS is absent, GALAHAD_NLS_solve will use reverse communication to obtain Hessian-vector products (see Section 2.7).

eval_SCALE is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the product $\mathbf{S}(\mathbf{x})\mathbf{v}$ of the user's preconditioner with a given vector $\mathbf{v}$. See Section 2.6.7 for details. If eval_SCALE is present, it must be declared EXTERNAL in the calling program. If eval_SCALE is absent, GALAHAD_NLS_solve will use reverse communication to obtain products with the preconditioner (see Section 2.7).

2.5.3 The termination subroutine

All previously allocated arrays are deallocated as follows:

```
CALL NLS_terminate( data, control, inform )
```

data is a scalar INTENT(INOUT) argument of type NLS_data_type exactly as for NLS_solve, which must not have been altered **by the user** since the last call to NLS_initialize. On exit, array components will have been deallocated.

control is a scalar INTENT(IN) argument of type NLS_control_type exactly as for NLS_solve.

inform is a scalar INTENT(OUT) argument of type NLS_inform_type exactly as for NLS_solve. The component inform%status will be set on exit, and a successful call to NLS_terminate is indicated when this component has the value 0. For other return values of inform%status, see Section 2.8.

2.6 Function and derivative values

2.6.1 The residual value via internal evaluation

If the argument eval_C is present when calling GALAHAD_NLS_solve, the user is expected to provide a subroutine of that name to evaluate the value of the residual functions $\mathbf{c}(\mathbf{x})$. The routine must be specified as

```
SUBROUTINE eval_C( status, X, userdata, c )
```

whose arguments are as follows:

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

status is a scalar INTENT(OUT) argument of type INTEGER(ip_), that should be set to 0 if the routine has been able to evaluate the residual functions and to a non-zero value if the evaluation has not been possible.

X is a rank-one INTENT(IN) array argument of type REAL(rp_) whose components contain the vector $\mathbf{x}$.

userdata is a scalar INTENT(INOUT) argument of type GALAHAD_userdata_type whose components may be used to communicate user-supplied data to and from the subroutines eval_C, eval_J, eval_H, eval_JPROD, eval_HPROD, eval_HPRODS and eval_SCALE (see Section 2.4.7).

C is a rank-one INTENT(OUT) array argument of type REAL(rp_) that should be set to the value of the residuals $\mathbf{c}(\mathbf{x})$ evaluated at the vector $\mathbf{x}$ input in X.

2.6.2 Jacobian values via internal evaluation

If the argument eval_J is present when calling GALAHAD_NLS_solve, the user is expected to provide a subroutine of that name to evaluate the values of the residual Jacobian $\mathbf{J}(\mathbf{x})$. The routine must be specified as

```
SUBROUTINE eval_J( status, X, userdata, J_val )
```

whose arguments are as follows:

status is a scalar INTENT(OUT) argument of type INTEGER(ip_), that should be set to 0 if the routine has been able to evaluate the residual Jacobian, and to a non-zero value if the evaluation has not been possible.

X is a rank-one INTENT(IN) array argument of type REAL(rp_) whose components contain the vector $\mathbf{x}$.

userdata is a scalar INTENT(INOUT) argument of type GALAHAD_userdata_type whose components may be used to communicate user-supplied data to and from the subroutines eval_C, eval_J, eval_H, eval_JPROD, eval_HPROD, eval_HPRODS and eval_SCALE (see Section 2.4.7).

J_val is a scalar INTENT(OUT) argument of type REAL(rp_), whose components should be set to the values of the Jacobian $\mathbf{J}(\mathbf{x})$ evaluated at the vector $\mathbf{x}$ input in X. The values should be input in the same order as that in which the array indices were given in nlp%J.

2.6.3 weighted residual Hessian values via internal evaluation

If the argument eval_H is present when calling GALAHAD_NLS_solve, the user is expected to provide a subroutine of that name to evaluate the values of the weighted Hessian $\mathbf{H}(\mathbf{x},\mathbf{y})$. The routine must be specified as

```
SUBROUTINE eval_H( status, X, Y, userdata, H_val )
```

whose arguments are as follows:

status is a scalar INTENT(OUT) argument of type INTEGER(ip_), that should be set to 0 if the routine has been able to evaluate the weighted Hessian and to a non-zero value if the evaluation has not been possible.

X is a rank-one INTENT(IN) array argument of type REAL(rp_) whose components contain the vector $\mathbf{x}$.

Y is a rank-one INTENT(IN) array argument of type REAL(rp_) whose components contain the vector $\mathbf{y}$.

userdata is a scalar INTENT(INOUT) argument of type GALAHAD_userdata_type whose components may be used to communicate user-supplied data to and from the subroutines eval_C, eval_J, eval_H, eval_JPROD, eval_HPROD, eval_HPRODS and eval_SCALE (see Section 2.4.7).

H_val is a scalar INTENT(OUT) argument of type REAL(rp_), whose components should be set to the values of the weighted Hessian $\mathbf{H}(\mathbf{x},\mathbf{y})$ evaluated at the vector $\mathbf{x}$ input in X and $\mathbf{y}$ input in Y The values should be input in the same order as that in which the array indices were given in nlp%H.

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

2.6.4 Jacobian-vector products via internal evaluation

If the argument eval_JPROD is present when calling GALAHAD_NLS_solve, the user is expected to provide a subroutine of that name to evaluate the sum $\mathbf{u}+\mathbf{J}(\mathbf{x})\mathbf{v}$ or $\mathbf{u}+\mathbf{J}^T(\mathbf{x})\mathbf{v}$ involving the product of the residual Jacobian $\mathbf{J}(\mathbf{x})$ or its transpose $\mathbf{J}^T(\mathbf{x})$ and a given vector $\mathbf{v}$. The routine must be specified as

```
SUBROUTINE eval_JPROD( status, X, userdata, transpose, U, V, got_j )
```

whose arguments are as follows:

status is a scalar INTENT(OUT) argument of type INTEGER(ip_), that should be set to 0 if the routine has been able to evaluate the sum $\mathbf{u}+\mathbf{J}(\mathbf{x})\mathbf{v}$ or $\mathbf{u}+\mathbf{J}^T(\mathbf{x})\mathbf{v}$ and to a non-zero value if the evaluation has not been possible.

X is a rank-one INTENT(IN) array argument of type REAL(rp_) whose components contain the vector $\mathbf{x}$.

userdata is a scalar INTENT(INOUT) argument of type GALAHAD_userdata_type whose components may be used to communicate user-supplied data to and from the subroutines eval_C, eval_J, eval_H, eval_JPROD, eval_HPROD, eval_HPRODS and eval_SCALE (see Section 2.4.7).

transpose is a scalar INTENT(IN) array argument of type default that will be set .TRUE. if the product involves the transpose of the Jacobian $\mathbf{J}^T(\mathbf{x})$ and .FALSE. if the product involves the Jacobian $\mathbf{J}(\mathbf{x})$ itself.

U is a rank-one INTENT(INOUT) array argument of type REAL(rp_) whose components on input contain the vector $\mathbf{u}$ and on output the sum $\mathbf{u}+\mathbf{J}(\mathbf{x})\mathbf{v}$ when %transpose is .FALSE. or $\mathbf{u}+\mathbf{J}^T(\mathbf{x})\mathbf{v}$ when %transpose is .TRUE..

V is a rank-one INTENT(IN) array argument of type REAL(rp_) whose components contain the vector $\mathbf{v}$.

got_j is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If the Jacobian has already been evaluated at the current $\mathbf{x}$ got_j will be PRESENT and set .TRUE.; if this is the first time the Jacobian is to be accessed at $\mathbf{x}$, either got_j will be absent or PRESENT and set .FALSE.. This gives the user the opportunity to reuse "start-up" computations required for the first instance of $\mathbf{x}$ to speed up subsequent products.

2.6.5 Hessian-vector products via internal evaluation

If the argument eval_HPROD is present when calling GALAHAD_NLS_solve, the user is expected to provide a subroutine of that name to evaluate the sum $\mathbf{u}+\mathbf{H}(\mathbf{x},\mathbf{y})\mathbf{v}$ involving the product of the weighted residual Hessian $\mathbf{H}(\mathbf{x},\mathbf{y})$ with a given vector $\mathbf{v}$. The routine must be specified as

```
SUBROUTINE eval_HPROD( status, X, Y, userdata, U, V, got_h )
```

whose arguments are as follows:

status is a scalar INTENT(OUT) argument of type INTEGER(ip_), that should be set to 0 if the routine has been able to evaluate the sum $\mathbf{u}+\mathbf{H}(\mathbf{x},\mathbf{y})\mathbf{v}$ and to a non-zero value if the evaluation has not been possible.

X is a rank-one INTENT(IN) array argument of type REAL(rp_) whose components contain the vector $\mathbf{x}$.

Y is a rank-one INTENT(IN) array argument of type REAL(rp_) whose components contain the vector $\mathbf{y}$.

userdata is a scalar INTENT(INOUT) argument of type GALAHAD_userdata_type whose components may be used to communicate user-supplied data to and from the subroutines eval_C, eval_J, eval_H, eval_JPROD, eval_HPROD, eval_HPRODS and eval_SCALE (see Section 2.4.7).

U is a rank-one INTENT(INOUT) array argument of type REAL(rp_) whose components on input contain the vector $\mathbf{u}$ and on output the sum $\mathbf{u}+\mathbf{H}(\mathbf{x},\mathbf{y})\mathbf{v}$.

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

V is a rank-one INTENT(IN) array argument of type REAL(rp_) whose components contain the vector $\mathbf{v}$.

got_h is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If the Hessian has already been evaluated at the current pair $(\mathbf{x},\mathbf{y})$ got_h will be PRESENT and set .TRUE.; if this is the first time the Hessian is to be accessed at $(\mathbf{x},\mathbf{y})$, either got_h will be absent or PRESENT and set .FALSE.. This gives the user the opportunity to reuse "start-up" computations required for the first instance of $(\mathbf{x},\mathbf{y})$ to speed up subsequent products.

2.6.6 Hessians-vector product matrix via internal evaluation

If the argument eval_HPRODS is present when calling GALAHAD_NLS_solve, the user is expected to provide a subroutine of that name to evaluate the values of the residual-Hessians-vector product matrix, $\mathbf{P}(\mathbf{x},\mathbf{v})$ at a given vector pair $(\mathbf{x},\mathbf{v})$. The routine must be specified as

```
SUBROUTINE eval_HPRODS( status, X, V, userdata, P_val, got_h )
```

whose arguments are as follows:

status is a scalar INTENT(OUT) argument of type INTEGER(ip_), that should be set to 0 if the routine has been able to evaluate the matrix $\mathbf{P}(\mathbf{x},\mathbf{v})$, and to a non-zero value if the evaluation has not been possible.

X is a rank-one INTENT(IN) array argument of type REAL(rp_) whose components contain the vector $\mathbf{x}$.

V is a rank-one INTENT(IN) array argument of type REAL(rp_) whose components contain the vector $\mathbf{v}$.

userdata is a scalar INTENT(INOUT) argument of type GALAHAD_userdata_type whose components may be used to communicate user-supplied data to and from the subroutines eval_C, eval_J, eval_H, eval_JPROD, eval_HPROD, eval_HPRODS and eval_SCALE (see Section 2.4.7).

P%val is a scalar INTENT(OUT) argument of type REAL(rp_), whose components should be set to the values of the Hessian residual-Hessians-vector product matrix, $\mathbf{P}(\mathbf{x},\mathbf{v})$, stored by columns and evaluated at the vector $\mathbf{x}$ input in X and $\mathbf{v}$ input in V. The values should be input in the same order as that in which the array indices were given in nlp%P.

got_h is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If the Hessians have already been evaluated at the current $\mathbf{x}$, got_h will be PRESENT and set .TRUE.; if this is the first time the Hessian is to be accessed at $\mathbf{x}$, either got_h will be absent or PRESENT and set .FALSE.. This gives the user the opportunity to reuse "start-up" computations required for the first instance of $\mathbf{x}$ to speed up subsequent products.

2.6.7 Preconditioner-vector products via internal evaluation

If the argument eval_SCALE is present when calling GALAHAD_NLS_solve, the user is expected to provide a subroutine of that name to evaluate the product $\mathbf{u} = \mathbf{S}(\mathbf{x})\mathbf{v}$ involving the user's preconditioner $\mathbf{S}(\mathbf{x})$ with a given vector $\mathbf{v}$. The symmetric matrix $\mathbf{S}(\mathbf{x})$ should ideally be chosen so that the eigenvalues of $\mathbf{S}(\mathbf{x})(\nabla_{xx}t_k(\mathbf{0}))^{-1}$ are clustered, where $t_k(\mathbf{s})$ is the current model of $f(\mathbf{x}+\mathbf{s})$. The routine must be specified as

```
SUBROUTINE eval_SCALE( status, X, userdata, U, V )
```

whose arguments are as follows:

status is a scalar INTENT(OUT) argument of type INTEGER(ip_), that should be set to 0 if the routine has been able to evaluate the product $\mathbf{S}(\mathbf{x})\mathbf{v}$ and to a non-zero value if the evaluation has not been possible.

X is a rank-one INTENT(IN) array argument of type REAL(rp_) whose components contain the vector $\mathbf{x}$.

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

userdata is a scalar INTENT(INOUT) argument of type GALAHAD_userdata_type whose components may be used to communicate user-supplied data to and from the subroutines eval_C, eval_J, eval_H and eval_SCALE (see Section 2.4.7).

U is a rank-one INTENT(OUT) array argument of type REAL(rp_) whose components on output should contain the product $\mathbf{u} = \mathbf{S}(\mathbf{x})\mathbf{v}$.

V is a rank-one INTENT(IN) array argument of type REAL(rp_) whose components contain the vector $\mathbf{v}$.

2.7 Reverse Communication Information

A positive value of inform%status on exit from NLS_solve indicates that GALAHAD_NLS_solve is seeking further information—this will happen if the user has chosen not to evaluate function or derivative values internally (see Section 2.6). The user should compute the required information and re-enter GALAHAD_NLS_solve with inform%status and all other arguments (except those specifically mentioned below) unchanged.

Possible values of inform%status and the information required are

2. The user should compute the vector of residual functions $\mathbf{c}(\mathbf{x})$ at the point $\mathbf{x}$ indicated in nlp%X. The required value should be set in nlp%C, and data%eval_status should be set to 0. If the user is unable to evaluate $\mathbf{c}(\mathbf{x})$—for instance, if one or more of the residual functions is undefined at $\mathbf{x}$—the user need not set nlp%C, but should then set data%eval_status to a non-zero value.

3. The user should compute the Jacobian matrix $\mathbf{J}(\mathbf{x})$ of the residuals $\mathbf{c}(\mathbf{x})$ at the point $\mathbf{x}$ indicated in nlp%X. The l-th component of the Jacobian stored according to the scheme input in the remainder of nlp%J (see Section 2.4.2) should be set in nlp%J%val(l), for $l = 1, \ldots,$ nlp%J%ne and data%eval_status should be set to 0. If the user is unable to evaluate a component of $\mathbf{J}(\mathbf{x})$—for instance, if a component of the Jacobian is undefined at $\mathbf{x}$—the user need not set nlp%J%val, but should then set data%eval_status to a non-zero value.

4. The user should compute the weighted Hessian $\mathbf{H}(\mathbf{x},\mathbf{y})$ at the point $\mathbf{x}$ given in nlp%X and vector $\mathbf{y}$ given in data%Y. The l-th component of the Hessian stored according to the scheme input in the remainder of nlp%H (see Section 2.4.2) should be set in nlp%H%val(l), for $l = 1, \ldots,$ nlp%H%ne and data%eval_status should be set to 0. If the user is unable to evaluate a component of $\mathbf{H}(\mathbf{x},\mathbf{y})$—for instance, if a component of the Hessian is undefined at $\mathbf{x}$—the user need not set nlp%H%val, but should then set data%eval_status to a non-zero value.

5. The user should compute ones of the sums $\mathbf{u}+\mathbf{J}(\mathbf{x})\mathbf{v}$ or $\mathbf{u}+\mathbf{J}^T(\mathbf{x})\mathbf{v}$ involving the product of the residual Jacobian $\mathbf{J}(\mathbf{x})$ or its transpose at the point $\mathbf{x}$, given in nlp%X, with a given vector $\mathbf{v}$. The vectors $\mathbf{u}$ and $\mathbf{v}$ are given in data%U and data%V respectively. If data%transpose is .FALSE., the resulting vector $\mathbf{u}+\mathbf{J}(\mathbf{x})\mathbf{v}$ should overwrite data%U and data%eval_status should be set to 0. Conversely if data%transpose is .TRUE., the resulting vector $\mathbf{u}+\mathbf{J}^T(\mathbf{x})\mathbf{v}$ should overwrite data%U and data%eval_status set to 0. If the user is unable to evaluate the sum—for instance, if a component of the Jacobian is undefined at $\mathbf{x}$—the user need not set data%U, but should then set data%eval_status to a non-zero value.

6. The user should compute the sum $\mathbf{u}+\mathbf{H}(\mathbf{x},\mathbf{y})\mathbf{v}$, involving the product of the weighted residual Hessian $\mathbf{H}(\mathbf{x},\mathbf{y})$ at the point $\mathbf{x}$ given in nlp%X and vector $\mathbf{y}$ given in data%Y with a given vector $\mathbf{v}$. The vectors $\mathbf{u}$ and $\mathbf{v}$ are given in data%U and data%V respectively, the resulting vector $\mathbf{u}+\mathbf{H}(\mathbf{x},\mathbf{y})\mathbf{v}$ should overwrite data%U and data%eval_status should be set to 0. If the user is unable to evaluate the sum—for instance, if a component of the Hessian is undefined at $\mathbf{x}$—the user need not set nlp%H%val, but should then set data%eval_status to a non-zero value.

7. The user should compute the residual-Hessians-vector-product matrix $\mathbf{P}(\mathbf{x},\mathbf{v})$, whose i-th column is the product $\mathbf{H}_i(\mathbf{x})\mathbf{v}$ between the Hessian $\mathbf{H}_i(\mathbf{x})$ of the i-th residual function $c_i(\mathbf{x})$ at the point $\mathbf{x}$ given in nlp%X and a given vector $\mathbf{v}$ specified in data%V. The nonzeros for column i must be stored in nlp%P%val(l), for l = nlp%P%ptr(i), ..., nlp%P%ptr(i+1) for each column i = 1,...,m, in the same order as the row indices

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

were assigned on input to nlp%P%row(l), and data%eval_status should be set to 0. If the user is unable to evaluate the matrix—for instance if a component of one of the Hessians is undefined at **x**—the user need not set nlp%P%val, but should then set data%eval_status to a non-zero value.

8. The user should compute the product $\mathbf{u} = \mathbf{S}(\mathbf{x})\mathbf{v}$ of their preconditioner $\mathbf{S}(\mathbf{x})$ at the point **x** indicated in nlp%X with the vector **v**. The vectors **v** is given in data%V, the resulting vector $\mathbf{u} = \mathbf{S}(\mathbf{x})\mathbf{v}$ should be set in data%U and data%eval_status should be set to 0. If the user is unable to evaluate the product—for instance, if a component of the preconditioner is undefined at **x**—the user need not set data%U, but should then set data%eval_status to a non-zero value.

2.8 Warning and error messages

A negative value of inform%status on exit from NLS_solve or NLS_terminate indicates that an error has occurred. No further calls should be made until the error has been corrected. Possible values are:

-1. An allocation error occurred. A message indicating the offending array is written on unit control%error, and the returned allocation status and a string containing the name of the offending array are held in inform%alloc_status and inform%bad_alloc, respectively.

-2. A deallocation error occurred. A message indicating the offending array is written on unit control%error and the returned allocation status and a string containing the name of the offending array are held in inform%alloc_status and inform%bad_alloc, respectively.

-3. The restriction nlp%n > 0 and nlp%m > 0, or requirements that nlp%J%type contains its relevant string 'DENSE', 'COORDINATE' or, 'SPARSE_BY_ROWS' and that nlp%H_type contains its 'DENSE', 'COORDINATE', 'SPARSE_BY_ROWS' has been violated.

-9. The analysis phase of the factorization failed; the return status from the factorization package is given in the component inform%factor_status.

-10. The factorization failed; the return status from the factorization package is given in the component inform%factor_status.

-11. The solution of a set of linear equations using factors from the factorization package failed; the return status from the factorization package is given in the component inform%factor_status.

-15. The preconditioner $\mathbf{S}(\mathbf{x})$ appears not to be positive definite.

-16. The problem is so ill-conditioned that further progress is impossible.

-17. The step is too small to make further impact.

-18. Too many iterations have been performed. This may happen if control%maxit is too small, but may also be symptomatic of a badly scaled problem.

-19. The elapsed CPU or system clock time limit has been reached. This may happen if either control%cpu_time_limit or control%clock_time_limit is too small, but may also be symptomatic of a badly scaled problem.

-82. The user has forced termination of GALAHAD_NLS_solve by removing the file named control%alive_file from unit unit control%alive_unit.

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

2.9 Further features

In this section, we describe an alternative means of setting control parameters, that is components of the variable control of type NLS_control_type (see Section 2.4.3), by reading an appropriate data specification file using the subroutine NLS_read_specfile. This facility is useful as it allows a user to change NLS control parameters without editing and recompiling programs that call NLS.

A specification file, or specfile, is a data file containing a number of "specification commands". Each command occurs on a separate line, and comprises a "keyword", which is a string (in a close-to-natural language) used to identify a control parameter, and an (optional) "value", which defines the value to be assigned to the given control parameter. All keywords and values are case insensitive, keywords may be preceded by one or more blanks but values must not contain blanks, and each value must be separated from its keyword by at least one blank. Values must not contain more than 30 characters, and each line of the specfile is limited to 80 characters, including the blanks separating keyword and value.

The portion of the specification file used by NLS_read_specfile must start with a "BEGIN NLS" command and end with an "END" command. The syntax of the specfile is thus defined as follows:

```
( .. lines ignored by NLS_read_specfile .. )
  BEGIN NLS
     keyword    value
     .......    .....
     keyword    value
  END
( .. lines ignored by NLS_read_specfile .. )
```

where keyword and value are two strings separated by (at least) one blank. The "BEGIN NLS" and "END" delimiter command lines may contain additional (trailing) strings so long as such strings are separated by one or more blanks, so that lines such as

```
BEGIN NLS SPECIFICATION
```

and

```
END NLS SPECIFICATION
```

are acceptable. Furthermore, between the "BEGIN NLS" and "END" delimiters, specification commands may occur in any order. Blank lines and lines whose first non-blank character is ! or * are ignored. The content of a line after a ! or * character is also ignored (as is the ! or * character itself). This provides an easy manner to "comment out" some specification commands, or to comment specific values of certain control parameters.

The value of a control parameters may be of three different types, namely integer, logical or real. Integer and real values may be expressed in any relevant Fortran integer and floating-point formats (respectively). Permitted values for logical parameters are "ON", "TRUE", ".TRUE.", "T", "YES", "Y", or "OFF", "NO", "N", "FALSE", ".FALSE." and "F". Empty values are also allowed for logical control parameters, and are interpreted as "TRUE".

The specification file must be open for input when NLS_read_specfile is called, and the associated device number passed to the routine in device (see below). Note that the corresponding file is REWINDed, which makes it possible to combine the specifications for more than one program/routine. For the same reason, the file is not closed by NLS_read_specfile.

2.9.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL NLS_read_specfile( control, device )
```

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

control is a scalar INTENT(INOUT) argument of type NLS_control_type (see Section 2.4.3). Default values should have already been set, perhaps by calling NLS_initialize. On exit, individual components of control and control%subproblem_control may have been changed according to the commands found in the specfile. Specfile commands and the component (see Section 2.4.3) of control and control%subproblem_control that each affects are given in Table 2.2.

command	component of control	value type
error-printout-device	%error	integer
printout-device	%out	integer
print-level	%print_level	integer
start-print	%start_print	integer
stop-print	%stop_print	integer
iterations-between-printing	%print_gap	integer
maximum-number-of-iterations	%maxit	integer
alive-device	%alive_unit	integer
jacobian-available	%jacobian_available	integer
hessian-available	%hessian_available	integer
history-length-for-non-monotone-descent	%non_monotone	integer
model-used	%model	integer
norm-used	%norm	integer
weight-update-strategy	weight_update_strategy	integer
absolute-residual-accuracy-required	%stop_c_absolute	real
relative-residual-reduction-required	%stop_c_relative	real
absolute-gradient-accuracy-required	%stop_g_absolute	real
relative-gradient-reduction-required	%stop_g_relative	real
minimum-relative-step-allowed	%stop_s	real
initial-regularization-weight	%initial_weight	real
minimum-regularization-weight	%minimum_weight	real
successful-iteration-tolerance	%eta_successful	real
very-successful-iteration-tolerance	%eta_very_successful	real
too-successful-iteration-tolerance	%eta_too_successful	real
regularization-weight-minimum-decrease-factor	%weight_decrease_min	real
regularization-weight-decrease-factor	%weight_decrease	real
regularization-weight-increase-factor	%weight_increase	real
regularization-weight-maximum-increase-factor	%weight_increase_max	real
minimum-objective-before-unbounded	%obj_unbounded	real
maximum-cpu-time-limit	%cpu_time_limit	real
maximum-clock-time-limit	%clock_time_limit	real
hessian-available	%hessian_available	logical
sub-problem-direct	%subproblem_direct	logical
retrospective-trust-region	%retrospective_trust_region	logical
renormalize-weight	%renormalize_weight	logical
space-critical	%space_critical	logical
deallocate-error-fatal	%deallocate_error_fatal	logical
alive-filename	%alive_file	character

Table 2.2: Specfile commands and associated components of control.

device is a scalar INTENT(IN) argument of type INTEGER(ip_), that must be set to the unit number on which the specfile has been opened. If device is not open, control will not be altered and execution will continue, but

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

an error message will be printed on unit control%error.

2.10 Information printed

If control%print_level is positive, information about the progress of the algorithm will be printed on unit control%out. If control%print_level $= 1$, a single line of output will be produced for each iteration of the process. This will include the values of the objective function and the norm of its gradient, the ratio of actual to predicted decrease following the step, the value of the regularization weight and the time taken so far. In addition, if a direct solution of the subproblem has been attempted, the Lagrange multiplier from the secular equation and the number of factorizations used will be recorded, while if an iterative solution has been used, the numbers of phase 1 and 2 iterations will be given.

If control%print_level ≥ 2 this output will be increased to provide significant detail of each iteration. This extra output includes residuals of the linear systems solved, and, for larger values of control%print_level, values of the variables and gradients. Further details concerning the attempted solution of the models may be obtained by increasing control%RQS_control%print_level and control%GLRT_control%print_level, while details about factorizations are available by increasing control%PSLS_control%print_level. See the specification sheets for the packages GALAHAD_GLRT, GALAHAD_PSLS and GALAHAD_RQS for details.

3 GENERAL INFORMATION

Use of common: None.

Workspace: Provided automatically by the module.

Other routines called directly: None.

Other modules used directly: NLS_solve calls the GALAHAD packages GALAHAD_CLOCK, GALAHAD_SYMBOLS, GALAHAD_NLPT, GALAHAD_SPECFILE, GALAHAD_PSLS, GALAHAD_GLRT, GALAHAD_RQS, GALAHAD_BSC, GALAHAD_SPACE, GALAHAD_ROOTS, GALAHAD_MOP, GALAHAD_NORMS, GALAHAD_STRING and GALAHAD_BLAS_interface.

Input/output: Output is under control of the arguments control%error, control%out and control%print_level.

Restrictions: nlp%n > 0, nlp%m > 0, nlp%J%type $\in$ {'DENSE','COORDINATE','SPARSE_BY_ROWS'} and nlp%H%type $\in$ {'DENSE', 'COORDINATE', 'SPARSE_BY_ROWS', 'DIAGONAL' }.

Portability: ISO Fortran 95 + TR 15581 or Fortran 2003. The package is thread-safe.

4 METHOD

An adaptive regularization method is used. In this, an improvement to a current estimate of the required minimizer, $\mathbf{x}_k$ is sought by computing a step $\mathbf{s}_k$. The step is chosen to approximately minimize a model $t_k(\mathbf{s})$ of $f_{\rho,r}(\mathbf{x}_k+\mathbf{s})$ that includes a weighted regularization term $\frac{\sigma_k}{p}\|s\|^p_{\mathbf{S}_k}$ for some specified positive weight σ_k. The quality of the resulting step $\mathbf{s}_k$ is assessed by computing the "ratio" $(f(\mathbf{x}_k)-f(\mathbf{x}_k+\mathbf{s}_k))/(t_k(\mathbf{0})-t_k(\mathbf{s}_k))$. The step is deemed to have succeeded if the ratio exceeds a given $\eta_s > 0$, and in this case $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$. Otherwise $\mathbf{x}_{k+1} = \mathbf{x}_k$, and the weight is increased by powers of a given increase factor up to a given limit. If the ratio is larger than $\eta_v \geq \eta_d$, the weight will be decreased by powers of a given decrease factor again up to a given limit. The method will terminate as soon as $f(\mathbf{x}_k)$ or $\|\nabla_x f(\mathbf{x}_k)\|$ is smaller than a specified value.

A choice of linear, quadratic or quartic models $t_k(\mathbf{s})$ is available (see §2.1), and normally a two-norm regularization will be used, but this may change if preconditioning is employed.

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

If linear or quadratic models are employed, an appropriate, approximate model minimizer is found using either a direct approach involving factorization of a shift of the model Hessian $\mathbf{B}_k$ or an iterative (conjugate-gradient/Lanczos) approach based on approximations to the required solution from a so-called Krlov subspace. The direct approach is based on the knowledge that the required solution satisfies the linear system of equations $(\mathbf{B}_k + \lambda_k \mathbf{I})\mathbf{s}_k = -\nabla_x f(\mathbf{x}_k)$ involving a scalar Lagrange multiplier λ_k. This multiplier is found by uni-variate root finding, using a safeguarded Newton-like process, by GALAHAD_RQS. The iterative approach uses GALAHAD_GLRT, and is best accelerated by preconditioning with good approximations to the Hessian of the model using GALAHAD_PSLS. The iterative approach has the advantage that only Hessian matrix-vector products are required, and thus the Hessian $\mathbf{B}_k$ is not required explicitly. However when factorizations of the Hessian are possible, the direct approach is often more efficient.

When a quartic model is used, the model is itself of least-squares form, and the package calls itself recursively to approximately minimize its model. The quartic model often gives a better approximation, but at the cost of more involved derivative requirements.

References:

The generic cubic regularization method is described in detail in

C. Cartis, N. I. M. Gould and Ph. L. Toint, "Adaptive cubic regularisation methods for unconstrained optimization. Part I: motivation, convergence and numerical results" *Mathematical Programming* **127(2)** (2011) 245–295,

and uses "tricks" as suggested in

N. I. M. Gould, M. Porcelli and Ph. L. Toint, "Updating the regularization parameter in the adaptive cubic regularization algorithm". *Computational Optimization and Applications* **53(1)** (2012) 1–22.

The specific methods employed here are discussed in

N. I. M. Gould, J. A. Scott and T. Rees, "Convergence and evaluation-complexity analysis of a regularized tensor-Newton method for solving nonlinear least-squares problems". *Computational Optimization and Applications* **73(1)** (2019) 1–35.

5 EXAMPLES OF USE

Suppose we wish to minimize the parametric objective function $\frac{1}{2}(x_1^2 x_3 + p)^2 + \frac{1}{2}(x_2^2 + x_3)^2$ when the parameter p takes the value 4. Starting from the initial guess $\mathbf{x} = (1,1,1)$, and noting that

$$\mathbf{c}(\mathbf{x}) = \begin{pmatrix} x_1^2 x_3 + p \\ x_2^2 + x_3 \end{pmatrix}, \mathbf{J}(\mathbf{x}) = \begin{pmatrix} 2x_1x_3 & 0 & x_1^2 \\ 0 & 2x_2 & 1 \end{pmatrix}, \mathbf{H}(\mathbf{x},\mathbf{y}) = \begin{pmatrix} 2x_3y_1 & 0 & 2x_1y_1 \\ 0 & 2y_2 & 0 \\ 2x_1y_1 & 0 & 0 \end{pmatrix} \& \mathbf{P}(\mathbf{x},\mathbf{v}) = \begin{pmatrix} 2x_3v_1 + 2x_1v_3 & 0 \\ 0 & 2v_2 \\ 2x_1v_1 & 0 \end{pmatrix},$$

we may use the following code:

```
PROGRAM GALAHAD_NLS_EXAMPLE !  GALAHAD 3.3 - 05/05/2021 AT 14:15 GMT
USE GALAHAD_NLS_double                     ! double precision version
IMPLICIT NONE
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )    ! set precision
TYPE ( NLPT_problem_type ):: nlp
TYPE ( NLS_control_type ) :: control
TYPE ( NLS_inform_type ) :: inform
TYPE ( NLS_data_type ) :: data
TYPE ( GALAHAD_userdata_type ) :: userdata
EXTERNAL :: EVALC, EVALJ, EVALH, EVALP
INTEGER :: s
INTEGER, PARAMETER :: m = 2, n = 3, j_ne = 4, h_ne = 3, p_ne = 3
REAL ( KIND = wp ), PARAMETER :: p = 4.0_wp  ! parameter p
```

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

```
! start problem data
   nlp%n = n ; nlp%m = m ; nlp%J%ne = j_ne ; nlp%H%ne = h_ne   ! dimensions
   ALLOCATE( nlp%X( n ), nlp%C( m ) )
   nlp%X = (/ 1.0_wp, 1.0_wp, 1.0_wp /)         ! start from (-1,1,1)
!  sparse co-ordinate storage format
   CALL SMT_put( nlp%J%type, 'COORDINATE', s )  ! Specify co-ordinate storage
   ALLOCATE( nlp%J%val( j_ne ), nlp%J%row( j_ne ), nlp%J%col( j_ne ) )
   nlp%J%row = (/ 1, 2, 1, 2 /)                 ! Jacobian J(x)
   nlp%J%col = (/ 1, 2, 3, 3 /)
   CALL SMT_put( nlp%H%type, 'COORDINATE', s )  ! Specify co-ordinate storage
   ALLOCATE( nlp%H%val( h_ne ), nlp%H%row( h_ne ), nlp%H%col( h_ne ) )
   nlp%H%row = (/ 1, 3, 2 /)                    ! Hessian H(x,y)
   nlp%H%col = (/ 1, 1, 2 /)                    ! NB lower triangle
   ALLOCATE( nlp%P%ptr( m + 1 ), nlp%P%row( p_ne ), nlp%P%val( p_ne ) )
   nlp%P%ptr = (/ 1, 3, 4 /)                    ! start of each column of P
   nlp%P%row = (/ 1, 3, 2 /)                    ! row indices of columns
   ALLOCATE( userdata%real( 1 ) )               ! Allocate space for parameter
   userdata%real( 1 ) = p                       ! Record parameter, p
! problem data complete ; solve using a Gauss-Newton model
   CALL NLS_initialize( data, control, inform ) ! Initialize control params
   control%subproblem_direct = .TRUE.           ! directly solve model problem
   control%model = 3                            ! Gauss-Newton model
   control%jacobian_available = 2               ! Jacobian is available
   inform%status = 1                            ! set for initial entry
   CALL NLS_solve( nlp, control, inform, data, userdata, eval_C = EVALC,       &
              eval_J = EVALJ, eval_H = EVALH )  ! Solve problem
   IF ( inform%status == 0 ) THEN               ! Successful return
     WRITE( 6, "( ' NLS: ', I0, ' iterations -',                               &
    &     ' optimal objective value =',                                        &
    &       ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" )                &
     inform%iter, inform%obj, nlp%X
   ELSE                                         ! Error returns
     WRITE( 6, "( ' NLS_solve exit status = ', I6 ) " ) inform%status
   END IF
! now solve using a Newton model
   control%subproblem_direct = .TRUE.           ! directly solve model problem
   control%model = 4                            ! Change to Newton model
   control%hessian_available = 2                ! Hessian is available
   nlp%X = (/ 1.0_wp, 1.0_wp, 1.0_wp /)         ! start from (-1,1,1)
   inform%status = 1                            ! set for initial entry
   CALL NLS_solve( nlp, control, inform, data, userdata, eval_C = EVALC,       &
              eval_J = EVALJ, eval_H = EVALH )  ! Solve problem
   IF ( inform%status == 0 ) THEN               ! Successful return
     WRITE( 6, "( ' NLS: ', I0, ' iterations -',                               &
    &     ' optimal objective value =',                                        &
    &       ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" )                &
     inform%iter, inform%obj, nlp%X
   ELSE                                         ! Error returns
     WRITE( 6, "( ' NLS_solve exit status = ', I6 ) " ) inform%status
   END IF
! finally solve using a tensor Gauss Newton model
   control%subproblem_direct = .TRUE.           ! directly solve model problem
   control%model = 6                            ! Change to tensor-GN model
   nlp%X = (/ 1.0_wp, 1.0_wp, 1.0_wp /)         ! start from (-1,1,1)
   inform%status = 1                            ! set for initial entry
```

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

```
CALL NLS_solve( nlp, control, inform, data, userdata, eval_C = EVALC,       &
                eval_J = EVALJ, eval_H = EVALH,                             &
                eval_HPRODS = EVALP )        ! Solve problem
IF ( inform%status == 0 ) THEN               ! Successful return
  WRITE( 6, "( ' NLS: ', I0, ' iterations -',                               &
 &     ' optimal objective value =',                                        &
 &       ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" )               &
  inform%iter, inform%obj, nlp%X
ELSE                                         ! Error returns
  WRITE( 6, "( ' NLS_solve exit status = ', I6 ) " ) inform%status
END IF
CALL NLS_terminate( data, control, inform )  ! delete internal workspace
DEALLOCATE( nlp%X, nlp%G, nlp%J%val, nlp%J%row, nlp%J%col, userdata%real )
DEALLOCATE( nlp%H%val, nlp%H%row, nlp%H%col )
DEALLOCATE( nlp%P%val, nlp%P%row, nlp%P%ptr )
END PROGRAM GALAHAD_NLS_EXAMPLE

SUBROUTINE EVALC( status, X, userdata, C )   ! residual
USE GALAHAD_USERDATA_double
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
INTEGER, INTENT( OUT ) :: status
REAL ( KIND = wp ), DIMENSION( : ),INTENT( IN ) :: X
REAL ( KIND = wp ), DIMENSION( : ),INTENT( OUT ) :: C
TYPE ( GALAHAD_userdata_type ), INTENT( INOUT ) :: userdata
REAL ( KIND = wp ) :: p
p = userdata%real( 1 )
C( 1 ) = X( 3 ) * X( 1 ) ** 2 + P
C( 2 ) = X( 2 ) ** 2 + X( 3 )
status = 0
RETURN
END SUBROUTINE EVALC

SUBROUTINE EVALJ( status, X, userdata, J_val )    ! Jacobian
USE GALAHAD_USERDATA_double
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
INTEGER, INTENT( OUT ) :: status
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X
REAL ( KIND = wp ), DIMENSION( : ), INTENT( OUT ) :: J_val
TYPE ( GALAHAD_userdata_type ), INTENT( INOUT ) :: userdata
REAL ( KIND = wp ) :: p
p = userdata%real( 1 )
J_val( 1 ) = 2.0_wp * X( 1 ) * X( 3 )
J_val( 2 ) = 2.0_wp * X( 2 )
J_val( 3 ) = X( 1 ) ** 2
J_val( 4 ) = 1.0_wp
status = 0
RETURN
END SUBROUTINE EVALJ

SUBROUTINE EVALH( status, X, Y, userdata, H_val ) ! scaled Hessian
USE GALAHAD_USERDATA_double
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
INTEGER, INTENT( OUT ) :: status
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X, Y
REAL ( KIND = wp ), DIMENSION( : ), INTENT( OUT ) :: H_val
```

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

```
   TYPE ( GALAHAD_userdata_type ), INTENT( INOUT ) :: userdata
   REAL ( KIND = wp ) :: p
   p = userdata%real( 1 )
   H_val( 1 ) = 2.0_wp * X( 3 ) * Y( 1 )
   H_val( 2 ) = 2.0_wp * X( 1 ) * Y( 1 )
   H_val( 3 ) = 2.0_wp * Y( 2 )
   status = 0
   RETURN
   END SUBROUTINE EVALH

   SUBROUTINE EVALP( status, X, V, userdata, P_val, got_h )
   USE GALAHAD_USERDATA_double
   INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
   INTEGER, INTENT( OUT ) :: status
   REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X
   REAL ( KIND = wp ), DIMENSION( : ), INTENT( INOUT ) :: P_val
   REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: V
   TYPE ( GALAHAD_userdata_type ), INTENT( INOUT ) :: userdata
   LOGICAL, OPTIONAL, INTENT( IN ) :: got_h
   P_val( 1 ) = 2.0_wp * ( X( 3 ) * V( 1 ) + X( 1 ) * V( 3 ) )
   P_val( 2 ) = 2.0_wp * X( 1 ) * V( 1 )
   P_val( 3 ) = 2.0_wp * V( 2 )
   status = 0
   RETURN
   END SUBROUTINE EVALP
```

Notice how the parameter p is passed to the function evaluation routines via the `real` component of the derived type `userdata`. The code produces the following output:

```
 NLS: 14 iterations - optimal objective value =  6.8988E-17
 Optimal solution =  -1.8704E+00  1.0693E+00 -1.1434E+00
 NLS: 12 iterations - optimal objective value =  7.7100E-18
 Optimal solution =   1.9304E+00  1.0361E+00 -1.0735E+00
 NLS: 6 iterations - optimal objective value =  4.6651E-13
 Optimal solution =   1.5102E+00  1.3243E+00 -1.7537E+00
```

If the Hessian is unavailable, but products of the form $\mathbf{u} + \mathbf{Hv}$ are, the same problem may be solved as follows:

```
   PROGRAM GALAHAD_NLS_EXAMPLE2 !  GALAHAD 3.3 - 05/05/2021 AT 14:15 GMT
   USE GALAHAD_NLS_double                     ! double precision version
   IMPLICIT NONE
   INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )    ! set precision
   TYPE ( NLPT_problem_type ):: nlp
   TYPE ( NLS_control_type ) :: control
   TYPE ( NLS_inform_type ) :: inform
   TYPE ( NLS_data_type ) :: data
   TYPE ( GALAHAD_userdata_type ) :: userdata
   EXTERNAL :: EVALC, EVALJ, EVALHPROD
   INTEGER :: s
   INTEGER, PARAMETER :: m = 2, n = 3, j_ne = 4, h_ne = 3, p_ne = 3
   REAL ( KIND = wp ), PARAMETER :: p = 4.0_wp  ! parameter p
! start problem data
   nlp%n = n ; nlp%m = m ; nlp%J%ne = j_ne ; nlp%H%ne = h_ne   ! dimensions
   ALLOCATE( nlp%X( n ), nlp%C( m ) )
   nlp%X = (/ 1.0_wp, 1.0_wp, 1.0_wp /)         ! start from (-1,1,1)
!  sparse co-ordinate storage format
   CALL SMT_put( nlp%J%type, 'COORDINATE', s )  ! Specify co-ordinate storage
```

All use is subject to the conditions of a BSD-3-Clause License.
See `http://galahad.rl.ac.uk/galahad-www/cou.html` **for full details.**

```
   ALLOCATE( nlp%J%val( j_ne ), nlp%J%row( j_ne ), nlp%J%col( j_ne ) )
   nlp%J%row = (/ 1, 2, 1, 2 /)                 ! Jacobian J(x)
   nlp%J%col = (/ 1, 2, 3, 3 /)
   ALLOCATE( userdata%real( 1 ) )               ! Allocate space for parameter
   userdata%real( 1 ) = p                       ! Record parameter, p
! problem data complete ; solve using a Newton model
   CALL NLS_initialize( data, control, inform ) ! Initialize control params
   control%jacobian_available = 2               ! Jacobian is available
   control%hessian_available = 1                ! only Hessian-vector products
   control%model = 4                            ! use the Newton model
   inform%status = 1                            ! set for initial entry
   CALL NLS_solve( nlp, control, inform, data, userdata, eval_C = EVALC,       &
              eval_J = EVALJ, eval_HPROD = EVALHPROD )  ! Solve problem
   IF ( inform%status == 0 ) THEN               ! Successful return
     WRITE( 6, "( ' NLS: ', I0, ' iterations -',                               &
    &     ' optimal objective value =',                                        &
    &       ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" )                &
     inform%iter, inform%obj, nlp%X
   ELSE                                         ! Error returns
     WRITE( 6, "( ' NLS_solve exit status = ', I6 ) " ) inform%status
   END IF
   CALL NLS_terminate( data, control, inform )  ! delete internal workspace
   DEALLOCATE( nlp%X, nlp%G, nlp%J%val, nlp%J%row, nlp%J%col, userdata%real )
   END PROGRAM GALAHAD_NLS_EXAMPLE2

   SUBROUTINE EVALC( status, X, userdata, C )   ! residual
   USE GALAHAD_USERDATA_double
   INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
   INTEGER, INTENT( OUT ) :: status
   REAL ( KIND = wp ), DIMENSION( : ),INTENT( IN ) :: X
   REAL ( KIND = wp ), DIMENSION( : ),INTENT( OUT ) :: C
   TYPE ( GALAHAD_userdata_type ), INTENT( INOUT ) :: userdata
   REAL ( KIND = wp ) :: p
   p = userdata%real( 1 )
   C( 1 ) = X( 3 ) * X( 1 ) ** 2 + P
   C( 2 ) = X( 2 ) ** 2 + X( 3 )
   status = 0
   RETURN
   END SUBROUTINE EVALC

   SUBROUTINE EVALJ( status, X, userdata, J_val )    ! Jacobian
   USE GALAHAD_USERDATA_double
   INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
   INTEGER, INTENT( OUT ) :: status
   REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X
   REAL ( KIND = wp ), DIMENSION( : ), INTENT( OUT ) :: J_val
   TYPE ( GALAHAD_userdata_type ), INTENT( INOUT ) :: userdata
   REAL ( KIND = wp ) :: p
   p = userdata%real( 1 )
   J_val( 1 ) = 2.0_wp * X( 1 ) * X( 3 )
   J_val( 2 ) = 2.0_wp * X( 2 )
   J_val( 3 ) = X( 1 ) ** 2
   J_val( 4 ) = 1.0_wp
   status = 0
   RETURN
```

All use is subject to the conditions of a BSD-3-Clause License.
See `http://galahad.rl.ac.uk/galahad-www/cou.html` **for full details.**

```
   END SUBROUTINE EVALJ

   SUBROUTINE EVALHPROD( status, X, Y, userdata, U, V, got_h ) ! Hessian product
   USE GALAHAD_USERDATA_double
   INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
   INTEGER, INTENT( OUT ) :: status
   REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X, Y
   REAL ( KIND = wp ), DIMENSION( : ), INTENT( INOUT ) :: U
   REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: V
   TYPE ( GALAHAD_userdata_type ), INTENT( INOUT ) :: userdata
   LOGICAL, OPTIONAL, INTENT( IN ) :: got_h
   U( 1 ) = U( 1 ) + 2.0_wp * Y( 1 ) * ( X( 3 ) * V( 1 ) + X( 1 ) * V( 3 ) )
   U( 2 ) = U( 2 ) + 2.0_wp * Y( 2 ) * V( 2 )
   U( 3 ) = U( 3 ) + 2.0_wp * Y( 1 ) * X( 1 ) * V( 1 )
   status = 0
   RETURN
   END SUBROUTINE EVALHPROD
```

Notice that storage for the Hessian is now not needed. This produces the following output:

```
 NLS: 12 iterations - optimal objective value =  3.4666E-18
 Optimal solution =   1.9304E+00  1.0361E+00 -1.0734E+00
```

If the user prefers to provide function and gradient information and Hessian-vector products without calls to specified routines, the following code is appropriate:

```
   PROGRAM GALAHAD_NLS_EXAMPLE3 !  GALAHAD 3.3 - 05/05/2021 AT 14:15 GMT
   USE GALAHAD_NLS_double                     ! double precision version
   IMPLICIT NONE
   INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )    ! set precision
   TYPE ( NLPT_problem_type ):: nlp
   TYPE ( NLS_control_type ) :: control
   TYPE ( NLS_inform_type ) :: inform
   TYPE ( NLS_data_type ) :: data
   TYPE ( GALAHAD_userdata_type ) :: userdata
   INTEGER :: s
   INTEGER, PARAMETER :: m = 2, n = 3, j_ne = 4, h_ne = 3, p_ne = 3
   REAL ( KIND = wp ), PARAMETER :: p = 4.0_wp  ! parameter p
! start problem data
   nlp%n = n ; nlp%m = m ; nlp%J%ne = j_ne ; nlp%H%ne = h_ne   ! dimensions
   ALLOCATE( nlp%X( n ), nlp%C( m ) )
   nlp%X = (/ 1.0_wp, 1.0_wp, 1.0_wp /)         ! start from (-1,1,1)
! problem data complete ; solve using a Newton model
   CALL NLS_initialize( data, control, inform ) ! Initialize control params
   control%jacobian_available = 1               ! only Jacobian-vector products
   control%hessian_available = 1                ! only Hessian-vector products
   control%model = 4                            ! use the Newton model
   inform%status = 1                            ! set for initial entry
10 CONTINUE
   CALL NLS_solve( nlp, control, inform, data, userdata )
   SELECT CASE ( inform%status ) !  is more information required?
   CASE ( 0 ) !  successful call
     WRITE( 6, "( ' NLS: ', I0, ' iterations -',                              &
    &     ' optimal objective value =',                                       &
    &       ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" )               &
     inform%iter, inform%obj, nlp%X
   CASE ( : - 1 ) !  unsuccessful call
```

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

```
    WRITE( 6, "( ' NLS_solve exit status = ', I6 ) " ) inform%status
  CASE ( 2 )
    nlp%C( 1 ) = nlp%X( 3 ) * nlp%X( 1 ) ** 2 + P
    nlp%C( 2 ) = nlp%X( 2 ) ** 2 + nlp%X( 3 )
    data%eval_status = 0
    GO TO 10  !  return to NLS_solve
  CASE ( 5 )
    IF ( data%transpose ) THEN
      data%U( 1 ) = data%U( 1 ) + 2.0_wp * nlp%X( 1 ) * nlp%X( 3 ) *          &
        data%V( 1 )
      data%U( 2 ) = data%U( 2 ) + 2.0_wp * nlp%X( 2 ) * data%V( 2 )
      data%U( 3 ) = data%U( 3 ) + data%V( 1 ) * nlp%X( 1 ) ** 2 + data%V( 2 )
    ELSE
      data%U( 1 ) = data%U( 1 ) + 2.0_wp * nlp%X( 1 ) * nlp%X( 3 ) *          &
        data%V( 1 ) + data%V( 3 ) * nlp%X( 1 ) ** 2
      data%U( 2 ) = data%U( 2 ) + 2.0_wp * nlp%X( 2 ) * data%V( 2 )           &
                                + data%V( 3 )
    END IF
    data%eval_status = 0
    GO TO 10  !  return to NLS_solve
  CASE ( 6 )
    data%U( 1 ) = data%U( 1 ) + 2.0_wp * data%Y( 1 ) *                        &
      ( nlp%X( 3 ) * data%V( 1 ) + nlp%X( 1 ) * data%V( 3 ) )
    data%U( 2 ) = data%U( 2 ) + 2.0_wp * data%Y( 2 ) * data%V( 2 )
    data%U( 3 ) = data%U( 3 ) + 2.0_wp * data%Y( 1 ) *                        &
       nlp%X( 1 ) * data%V( 1 )
    data%eval_status = 0
    GO TO 10  !  return to NLS_solve
  END SELECT
  CALL NLS_terminate( data, control, inform )  ! delete internal workspace
  DEALLOCATE( nlp%X, nlp%G )
  END PROGRAM GALAHAD_NLS_EXAMPLE3
```

This produces the same output as in the previous case.

All use is subject to the conditions of a BSD-3-Clause License.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**